UNIVERSITY
OF TURKU

ACDWH

A patented method for active data warehousing

Jari Myllylahti

University of Turku

Faculty of Technology
Department of Computing
Computer Science
Licentiate programme

Supervised by

Assistant Professor, Tuomas Mäkilä
University of Turku

Professor Emeritus, Olli Nevalainen
University of Turku

Reviewed by

Professor, Timo Knuutila
University of Turku

Professor, Jyrki Nummenmaa
University of Tampere

The originality of this publication has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

ISBN 978-951-29-9640-7

Dedicated to

my wife Virpi. Who makes my days filled with love. I love you.

"She will listen to me
When I want to speak
About the world we live in
And life in general"

my son Iiro. Who has inherited my bad sense of humor and who is by far superior to me in academic capabilities. Let this be an example of what you can accomplish yourself.

"To the infinity, and beyond!"

Clones, my brothers in scene. You know who you are.

"See the stars and see the rainbows - see it all
Don't turn away - you're all you've got
In a faceless world
You can run - but you can't hide
It's yourself who waits for you inside"

010101100110000100100000011110100111001001111010
011000100110010101101100001000000110001001110011
001000000100111001100001011100010110010101110010
011010100010000001010111011000100111010101100001
0010000001010011011111001011100100110011101110000
011101010111001001100101001011000010000000110001
001110010011011000110001001000001110001010000000
100100110010000000110010001100000011001000110010

UNIVERSITY OF TURKU
Faculty of Technology
Department of Computing
Computer Science
JARI MYLLYLAHTI: AcDWH – A patented method for active data warehousing
Licentiate thesis, 81 pages + appendices.
January 2024

ABSTRACT

The traditional needs of data warehousing from monthly, weekly or nightly batch processing have evolved to near real-time refreshment cycles of the data, called active data warehousing. While the traditional data warehousing methods have been used to batch load large sets of data in the past, the business need for extremely fresh data in the data warehouse has increased. Previous studies have reviewed different aspects of the process along with the different methods to process data in data warehouses in near real-time fashion. To date, there has been little research of using partitioned staging tables within relational databases, combined with a crafted metadata driven system and parallelized loading processes for active data warehousing.

This study provides a throughout description and suitability assessment of the patented AcDWH method for active data warehousing. In addition, this study provides a review and a summary of existing research on the data warehousing area from the era of start of data warehousing in the 1990's to the year 2020. The review focuses on different parts of the data warehousing process and highlights the differences compared to the AcDWH method. Related to the AcDWH, the usage of partitioned staging tables within a relational database in combination of meta data structures used to manage the system is discussed in detail. In addition, two real-life applications are disclosed and discussed on high level. Potential future extensions to the methodology are discussed, and briefly summarized.

The results indicate that the utilization of AcDWH method using parallelized loading pipelines and partitioned staging tables can provide enhanced throughput in the data warehouse loading processes. This is a clear improvement on the study's field. Previous studies have not been considering using partitioned staging tables in conjunction with the loading processes and pipeline parallelization. Review of existing literature against the AcDWH method together with trial and error -approach show that the results and conclusions of this study are genuine.

The results of this study confirm the fact that also technical level inventions within the data warehousing processes have significant contribution to the advance of methodologies. Compared to the previous studies in the field, this study suggests a simple yet novel method to achieve near real-time capabilities in active data warehousing.

KEYWORDS: active data warehousing, real-time, partitioning, staging

TURUN YLIOPISTO
Teknillinen Tiedekunta
Tietotekniikan Laitos
Tietojenkäsittelytiede
JARI MYLLYLAHTI: AcDWH – Patentoitu menetelmä aktiiviseen tietovarastointiin
Lisensiaatin tutkielma, 81 s + liitteet.
tammikuu 2024

TIIVISTELMÄ

Perinteiset tarpeet tietovarastoinnille kuukausittaisen, viikoittaisen tai yöllisen käsittelyn osalta ovat kehittyneet lähes reaaliaikaista päivitystä vaativaksi aktiiviseksi tietovarastoinniksi. Vaikka perinteisiä menetelmiä on käytetty suurten tietomäärien lataukseen menneisyydessä, liiketoiminnan tarve erittäin ajantasaiselle tiedolle tietovarastoissa on kasvanut. Aikaisemmat tutkimukset ovat tarkastelleet erilaisia prosessin osa-alueita sekä erilaisia menetelmiä tietojen käsittelyyn lähes reaaliaikaisissa tietovarastoissa. Tutkimus partitioitujen relaatiotietokantojen väliaikaistaulujen käytöstä aktiivisessa tietovarastoinnissa yhdessä räätälöidyn metatieto-ohjatun järjestelmän ja rinnakkaislatauksen kanssa on ollut kuitenkin vähäistä.

Tämä tutkielma tarjoaa kattavan kuvauksen sekä arvioinnin patentoidun AcDWH-menetelmän käytöstä aktiivisessa tietovarastoinnissa. Työ sisältää katsauksen ja yhteenvedon olemassa olevaan tutkimukseen tietovarastoinnin alueella 1990-luvun alusta vuoteen 2020. Kirjallisuuskatsaus keskittyy eri osa-alueisiin tietovarastointiprosessissa ja havainnollistaa eroja verrattuna AcDWH-menetelmään. AcDWH-menetelmän osalta käsitellään partitioitujen väliaikaistaulujen käyttöä relaatiotietokannassa, yhdessä järjestelmän hallitsemiseen käytettyjen metatietorakenteiden kanssa. Lisäksi kahden reaalielämän järjestelmän sovellukset kuvataan korkealla tasolla. Tutkimuksessa käsitellään myös menetelmän mahdollisia tulevia laajennuksia menetelmään tiivistetysti.

Tulokset osoittavat, että AcDWH-menetelmän käyttö rinnakkaisilla latausputkilla ja partitioitujen välitaulujen käytöllä tarjoaa tehokkaan tietovaraston latausprosessin. Tämä on selvä parannus aikaisempaan tutkimukseen verrattuna. Aikaisemmassa tutkimuksessa ei ole käsitelty partitioitujen väliaikaistaulujen käyttöä ja soveltamista latausprosessin rinnakkaistamisessa.

Tämän tutkimuksen tulokset vahvistavat, että myös tekniset keksinnöt tietovarastointiprosesseissa ovat merkittävässä roolissa menetelmien kehittymisessä. Aikaisempaan alan tutkimukseen verrattuna tämä tutkimus ehdottaa yksinkertaista mutta uutta menetelmää lähes reaaliaikaisten ominaisuuksien saavuttamiseksi aktiivisessa tietovarastoinnissa.

ASIASANAT: Tietovarastointi, reaaliaikaisuus, partitiointi, väliaikaistaulut

Table of Contents

Acknowledgements

I would like to express my profound gratitude to Professor Emeritus Olli Nevalainen for his invaluable guidance and support throughout my academic journey. His mentorship for the past 30 years has been instrumental in fostering my intellectual growth and also in shaping this research. His insightful comments and constructive, detailed feedback have continuously pushed me to enhance my thesis and develop a more comprehensive analysis. Olli's expertise, assertiveness, commitment, and dedication have always inspired me. I am indebted to him for his continuous encouragement and discussions that have facilitated the advancement of this thesis. I am forever grateful for Olli's patience with me over the past quarter of a century towards this very day. I am honored to have had Olli by my side on this project throughout the journey.

Furthermore, I would like to acknowledge Assistant Professor Tuomas Mäkilä's support for my work during the last years of my research journey. Despite his busy schedule he made time to discuss with me, providing valuable guidance and advice. His responses to my inquiries and his willingness to share his expertise have been greatly appreciated.

I would also like to extend my sincere appreciation to my employer, Tietoevry Corporation, for granting the original support for my post graduate studies and allowing me to focus on my thesis as needed.

Last but surely not least, I would like to express my heartfelt gratitude to my wife Virpi and my son Iiro, whose belief in my abilities has been a constant source of motivation. Their love, understanding, and support have been the fuel to finalize this thesis. I am forever grateful for their enduring faith in my ability to accomplish this.

Helsinki
January 15, 2024
Jari Myllylahti

Abbreviations

BI	Business Intelligence. BI means the concept of providing analytic systems for business users. These systems provide insights on specific business-related questions. The BI systems are usually constructed as DWHs.
CPU	Central Processing Unit. CPU is the processor of a computer.
DM	Data mart. DM is a lightly summarized area and structure in a DWH that contains summarized data on a specific subject area for example for departmental usage. DM uses typically a specific data model for reporting and analysis which called a star schema. This data model is focused to deliver fast reporting and analysis on one subject area, for example customer data or customer purchase transactions in a shop.
DSS	Decision Support Systems. DSS provide analytical view to a specific line of business or a company. These systems are used to support decision making in companies or lines of business. DSS typically presents or consumes data from a DWH.
DWH	Data Warehousing or Data Warehouse. DWH is a method and database structure where operational system data are replicated into a DWH database structure. This structure holds historical and current atomic data usually stored in a normalized form. The methodology is further described in Chapters 1 and 2.
ELT	Extract-Load-Transform. ELT changes the approach of ETL in manner, that first two process elements of ELT (Extract and Load) are executed before transform. Transform phase is executed within the database after whereas ETL's Transform part of the process is executed outside of the database. This change of process for loading the source system data to the DWH is made to utilize databases system's functionality, scalability, and efficiency for the Transform operations.

ETL	Extract-Transform-Load. The ETL process is typically used within DWH environments to process data. Extract part of the process extracts the data from source system(s), Transform part executes various data transformations within the process and Load part loads the data to DWH. ETL processes are typically constructed with a specific ETL tool instead of programming the processes by yourself. There are multiple technology products in the market for ETL. Most of them are separately installed from the database systems, but some (like Oracle Warehouse Builder) are installed within database systems.
GUI	Graphical User Interface. GUI is an interface that gives the ability for users to interact with computers through graphical icons instead of text-based user interfaces.
I/O	Input / Output. I/O means input and output peripherals, such as keyboards, displays, disk and tape devices.
ODS	Operational Data Store. ODS is an operational database usually integrating data from multiple source systems. It is designed to support reporting from operational data and offloading the reporting workloads from the operational systems. ODS also implements data integration and usually also data cleansing, redundancy removal and data integrity checking.

List of Original Publications

This thesis and the proposed AcDWH method are based on the following original publication, which is referred to in the text by the Roman numeral:

I Jari Myllylahti. European Patent Specification EP 1 959 359 B1. *European Patent Bulletin*, 2017; issue 47: 30 pages.

1 Introduction

While data warehousing (DWH) models have been utilized for over two decades for Decision Support Systems (DSS), analytics and business intelligence, yet there has not been really a drive for enhanced techniques for near real-time delivery and access of the data. The shift towards more refreshed data in business intelligence has been the driver to implement more fresh state of the business intelligence platforms in the form of improved the data warehouses.

The existing DWH models rely heavily on bulk loading techniques and the loading of data from source systems to DWH takes place typically during the night and the load contains data from one day. These techniques are sufficient when the transaction volumes are not too big and when there is no real business driver to access the data more frequently than the view of the previous day. These models have been utilized from the very beginning of the DWH era and they are still applied in large parts of the world's DWH environments.

Today's business drivers demand more fresh data, for which these typical nightly load windows are not sufficient. Typical need is a few minutes gap in between the refreshment cycles of the data. For these environments the traditional way of refreshing DWH data during the night is not sufficient anymore.

In this thesis we study and present how a partitioned staging table can be utilized in active DWH environments. The study question is if the partitioned staging table in combination with parallelized loading processes to and from the staging table can help to enhance active DWH systems. The study also presents enhancements to DWH system's throughput and manageability. The study reviews existing research and summarizes their key findings. Differences between existing research and the studied partitioned staging table and parallelized loading processes are highlighted.

The thesis presents a patented method of a data management system (AcDWH) which is an optimized and novel method for implementing active DWH systems. The method allows parallel asynchronous access to the data being delivered to the DWH. The AcDWH method divides the incoming data into buckets and the data flow management system is based on handling these buckets. The method consists of an arbitrary number of asynchronous data provider and delivery processes.

The AcDWH processes deliver and distribute the data through a common metadata layer of control data. The metadata tables log different phases of the process and provide a safe mechanism for concurrency and data delivery and efficiency control. The efficiency of the processes can be controlled and throttled through the administrative metadata layer so that the process can adjust itself to the needs of the business and to the capacity of the platform it runs on.

The thesis also presents the foundational principles on physical access of the data. The process can be parallelized and scaled, and different parts of the process can be isolated from each other by utilizing this specific method. This way the process can achieve high grade of parallelism, throughput and near real-time freshness of the data on the DWH.

For the scope of this thesis, the focus will be of the staging table physical structures and on the patented data management system. In addition, the focus will be on the AcDWH processes handling the processing pipelines. The method of applying parallelism to the different parts of processing pipeline will be discussed and addressed separately, providing examples on principles and correlation to real world problems.

In chapter 2, the background of DWH is discussed. Chapter 3 discusses problems identified in active DWH with traditional techniques. Chapter 4 gives a review on studies and literature about DWH and the staging area processing. A novel method for active DWH and data management system, AcDWH, is described in chapter 5 and its different parts are discussed in detail. Chapter 6 gives two examples of real-life implementations of AcDWH system. In chapter 7 extension possibilities to AcDWH are discussed and chapter 8 shows results and discusses the advantages of AcDWH over traditional DWH methods. Chapter 9 concludes with a summary of the thesis.

2 Data Warehousing

In this chapter the background of the data warehousing (DWH) systems is discussed. The chapter describes the typical structures and methodologies used in DWH solutions. The chapter also describes different parts of a typical DWH process and addresses different indexing techniques used in these systems.

2.1 Background

A DWH is a system with techniques & methodologies for managing data from different sources and combining them into a single DWH to provide insights to relevant business questions. A DWH is created from multiple components which aid the use of the specific data for strategic purposes. A DWH provides a database and system design which helps to keep historic details of the subject area of the DWH, to reduce the response time and to enhance the performance of queries for reports and analytics from it. [33]

A DWH consists of large amounts of data which are designed and organized for both historic data queries and strategic analyses in contradiction to transaction processing systems. DWH processes incoming source system data into business information and makes it available to business and analytical users.

The database of the DWH is separate from the organization's operational systems. The DWH is an environment and a database which consists of combined information from the source systems. It is a constructed system which provides current and historical decision support information to business users. The previous is typically cumbersome to access and present using operational databases and systems.

For example, a report on the financial system information on previous company fiscal year can easily include tens of join conditions and tables. These types of queries will slow down the response time of the query and report on the operational system. These queries will also have effect on the throughput of other database operations on the operational system while the queries are run.

In DWH the database design and structure are separated into two areas; to the actual DWH structures which are usually in normalized form and to separate structures supporting subject area queries, so-called data marts (DM) [1,4]. The latter structures are built with dimensional modelling and star schema. Within the star schema the *fact tables* are central to the design, holding all the relevant data for a data item. *Dimension tables* are holding all the relevant data for the dimension [1,4]. In this financial system example, the Supplier invoices table is the fact table, and the Supplier table is the dimension table. Figure 1 shows the generic architecture of a DWH system.

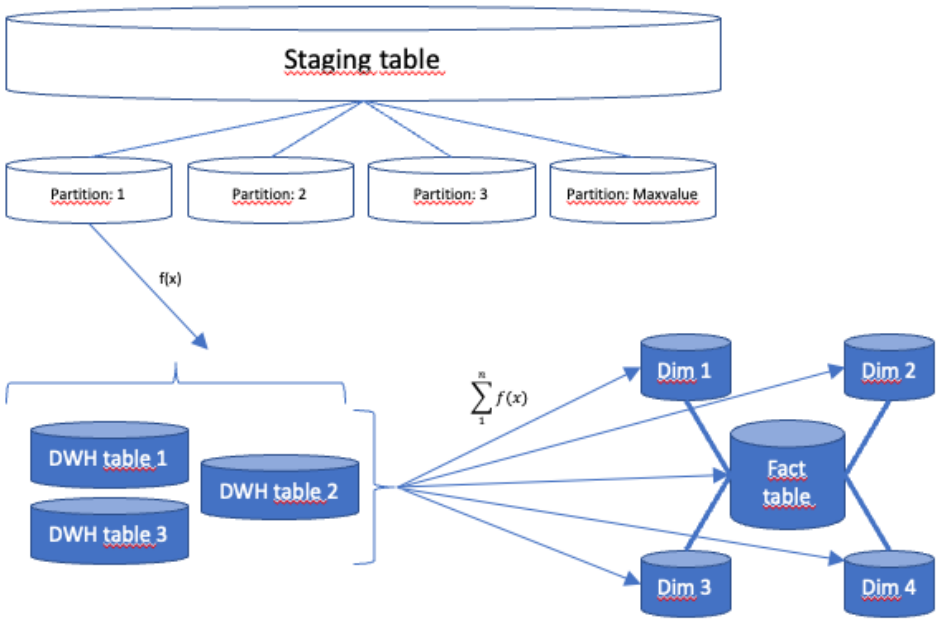

Figure 1. **Generic architecture of a DWH.**

The fact table has a column for each detail. The facts are usually numeric values that can provide business with aggregate views, for example providing a monetary sum of all invoices. A dimension is a specific attribute to the fact. Dimensions are valuable items to the business, such as supplier, invoicing month, invoicing country and so forth.

While the data is loaded into the fact table, the dimension attributes of a fact table row are replaced by a surrogate key pointing to the dimension table. The dimension table holds all the relevant details of a dimension record. An example being customer dimension, where the record has attributes such as name, address, social security number and other relevant attributes.

As an example, if the customer dimension record exists, the surrogate is fetched from the dimension table by searching for the customer number or name, and the surrogate key (customer id for example) is inserted into the fact table column giving a reference to the right entity on the dimension table. The same would apply for all the relevant dimensional attributes, that are connected with the relevant fact.

Continuing the example, if the dimension record is not existing in the dimension table, a new record is inserted into the dimension table. A new customer id is generated, and the applicable dimension details are updated to the record from the source systems. A simple star schema structure is illustrated in Figure 2.

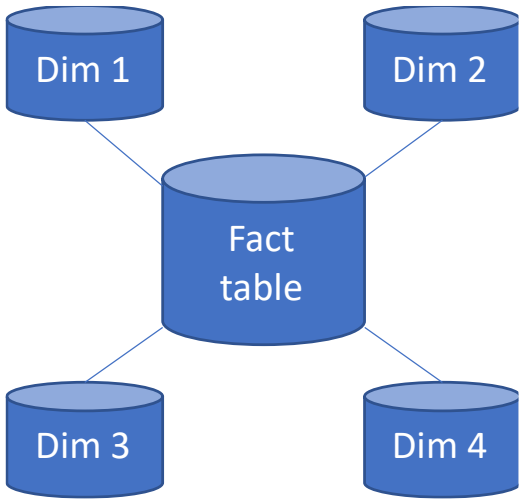

Figure 2. Star schema used in a DWH.

What is typical to the DWH, is that the dimension records are fetched from the operational source system databases on a specified interval (such as once per day during night) prior to the actual loading of the fact data. This causes the dimension records being up to date when the fact table(s) are loaded from the source systems and there is no need to generate the records during the fact table loading.

The afore mentioned star schema requires transformation of data while loading it into the DWH. This has an implication of having to use an intermediate storage table to ease the processing and to minimize the resource wear and burden on the source systems. These intermediate storage tables are called *staging tables* [4]. The source system data is loaded into these staging tables, from where the Extract, Transformation and Load (ETL) processes move the data further into the applicable star schema tables for analytical and query use.

2.2 Structure of the traditional DWH method

The traditional DWH processes are controlled and run on either monthly, weekly or daily basis. Figure 3 shows a high-level model of the traditional DWH processing.

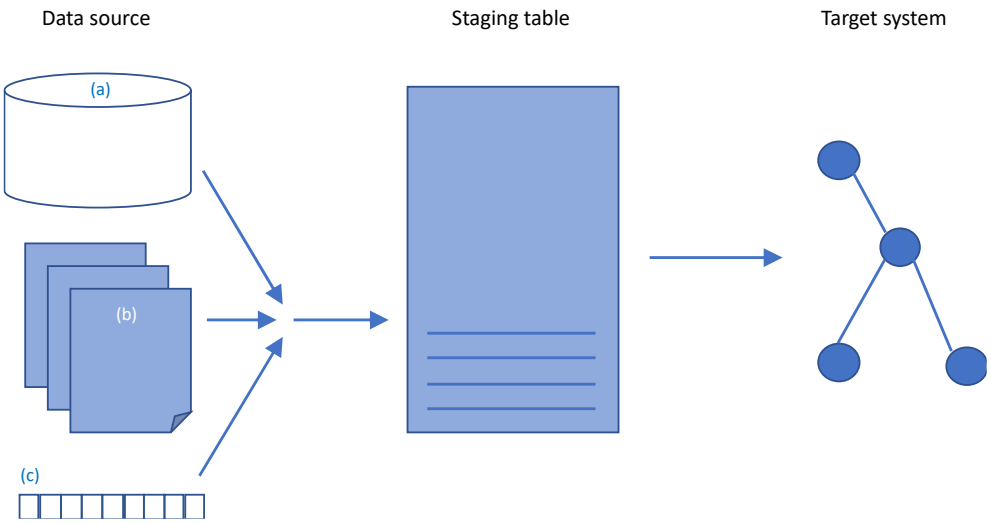

Figure 3. High level process description of traditional DWH method using sources such as databases (a), flat files (b) or message queuing systems (c).

The data sources are read on daily basis and the data is delivered to the staging area (e.g. a staging table) for further processing. The staging table is read and the whole data is delivered to the target system(s) for further data analysis.

Publications on DWH discuss the design topics and relevant schematics for this setup. These techniques are widely in use in traditional data warehousing environments. A literature review is presented in chapter 4.

The reading of the source data takes extensive time to finish as the data from a whole day, or even a longer period of time, is processed at once and typically during night. The data is not transformed at this stage, and it is written to the staging table exactly in the same format it is on the source system. Erroneous records are written to error logging structures for possible correcting and reprocessing of the data.

The staging table is read in the next part of the process to deliver the data into data warehousing structures. Loading the data to the data warehousing structures is done by reading data from staging table structures and transforming the data to normalized model in the DWH. This transforming phase of the landing process is the most resource consuming part of this process.

What is typical of the DWH is that these structures are in normalized form of data base schema and provide a solid layer for tracking changes over time.

The data is delivered to a reporting layer (e.g., a star schema consisting of a fact table and dimension tables) after the data has been processed in the DWH layer. This way the data can be analyzed by viewing it in different dimensions. Dimensional modelling provides the business users a simple yet powerful way to browse around their analytical data and analyze it on different aspects, i.e. one can e.g. make a summary of data for a given client over a period of time. The data loading of the star schema takes place once a day, aggregating the data on selected dimensions.

The main phases of the traditional DWH ETL process are:

1. Read the data from source system(s) and deliver it into a staging area in the DWH
2. Transform the data to the format of the target model
3. Load the transformed data into the target DWH and reporting models
4. Rebuild indexing structures that support the reporting models

DWH systems are Decision Support Systems (DSS) by definition and they provide an analytical view of data on aggregated and grouped level at predefined intervals [33]. In the case the business questions do not need to be answered more frequent than daily and there is no need to report on transactional basis, then the refreshment of the aggregated data into the object system can happen also on the same frequency.

Phase 1 typically inputs the data in from previous day. This part of the process delivers the data usually in the same format or in a mixture of the source and target formats. This way the transformation of the data can be traced back to the staging area in the case there is something wrong with the processing. This phase is time consuming as large amount of data is read from sources and the reading is typically limited to a small set of reading methods. These methods address the source data in a similar way to source applications. One of the problems on the traditional approach is that this phase cannot be parallelized which may create a bottleneck on the process.

Phase 2 transforms the data into the format of target model and attaches surrogate keys for the dimension objects to the table rows. This phase also generates new entities to dimension tables as the new dimension data is transferred from the source systems. This phase typically involves heavy calculations, aggregations and exotic transformations of different kinds. This part of the ETL process takes most of the time and resources, as the amount of records which will be addressed may be massive.

Phase 3 delivers (e.g., loads) the data into the target system, the DWH. The writing can and most often will be done by utilizing bulk loading mechanisms, delivering huge blocks of data directly to the database engine to make the load in the most efficient way. At the end of the phase the index structures must be rebuilt to facilitate the reporting on the data model. This part takes a lot of time because the complex indexing setup is typically heavy for reporting structures and index creation takes both CPU time and I/O resources.

2.3 Indexing techniques for DWH

DWH systems are usually indexed with b-tree and bitmap indices like operational databases. The indexing techniques rely on standard methods and this is valid regardless of the data volumes in the DWH. Some DWH environments utilize table and index partitioning to manage great data volumes. Instead of using global non-partitioned indices DWH systems are also using local indices which are partitioned according to underlying table partitions. This method will help to remove the congestion on the table and indices and enhance the throughput of different database operations [34].

In addition to the DWH structures mentioned above there are specific indexing techniques for the star schema to support reporting. A normal way to index a star schema is to index all dimension surrogate key attributes in the fact table with bitmap indices. In addition, some of the required search fields for analysis can be indexed for faster searches and aggregation. At the end of this phase the staging table is truncated as the records are processed, thus making the table available for the next load.

The difference between normal b-tree and bitmap indices is their internal structure, while normal indices are arranged to a b-tree structure the bitmap indexes are arranged in a two-dimensional binary array. The difference in their behavior is that bitmap indices are extremely usable and fast in addition to low space consumption in low cardinality columns. What makes bitmap indices well superior to b-tree indices is when two or more bitmap indexed columns can be used in search criteria. Then the database engine can merge the bitmap indices and generate the result set extremely fast [34].

3 Challenges in DWH

This chapter addresses the challenges in traditional data warehousing (DWH) systems and approaches. The challenges have been identified both by existing research and literature, as well as by experience while constructing the patented AcDWH method.

3.1 General problems on traditional DWH methods

The traditional DWH methods incorporate number of drawbacks in terms of

1. efficiency,
2. repeatability, and
3. efficient concurrent read and write access to DWH structures.

Initially, the data extracting phase is limited to very narrow reading of source systems. When the amount of data is large and the reading mechanisms are limited to typical reading patterns of the source systems, the outcome cannot be excellent. The source systems cannot be modified to address the needs of the DWH process as the systems have their own transactional needs to be fulfilled. The source systems have been designed for facilitating only their own data accessing needs, anything else is irrelevant.

Secondly, the data transformation to the target format takes time and resources. As the volume of transformed data is huge, the processing will need large amounts of memory and CPU time. The efficiency of the transformation is highly dependent on the resources of the transformation platform. If the platform is running short on the memory and CPU resources, the efficiency and throughput of the DWH system will deteriorate.

Similarly, the loading process depends on the amount of the data to be loaded. Whenever the data is loaded in, the loading process depends on the I/O capacity of the target DWH platform. No matter how performant the target platform is, the loading of a day's data will take time. This is the case particularly for the maintenance of indexing structures for the reporting model. These structures must be dropped before the data will be loaded in. If the indices would be up during the load, the impact on performance would be enormous while the indices would be kept up to date during the loading process. Currently, the index structures are typically dropped and then rebuilt after the data loading has been done. As the data volume loaded is huge, the rebuilding of the index structures is time consuming.

To sum up, traditional DWH techniques suffer from several specific efficiency problems:

1. The source system reading cannot be streamlined, parallelized and configured in a manner where the source system would facilitate for both the source system application and also the analysis of the full set of data for data warehousing purposes. The reading is a time and resource consuming process on huge volumes of data.
2. The same problems are present also in the transformation phase. As the amount of data is large, transforming the data to the needed format for the analysis will consume resources for the transformation engine. It cannot be avoided.
3. The loading process efficiency is depending on the data volume loaded. Populating large number of records and rebuilding index structures will take a significant amount of time. This will make the solution useless to analytical use case, with regards to the near real time requirements.
4. The space consumption of the staging table is remarkable. In a typical method the data is read from the source systems once per night, and the data is transferred to the data warehousing structures with large batch jobs. Utilizing this kind of structure within a near real-time solution would cause the staging table to grow unmanaged, causing the system to either halt or slow down drastically after an arbitrary amount of executions.

3.2 The high watermark problem on traditional DWH staging tables

The traditional staging tables are handled in a distinct way. The traditional DWH process will load a standard interval portion of data into the staging table; typical interval being either once per day, week or month.

Given the characteristics of such a construct, the loading of data into the staging table is extremely straightforward. On the other hand, the method will cause different issues on the efficiency of the further loading mechanism from the staging table.

The traditional method forces the system to process the whole staging table at a time if no additional load batch identification mechanisms are constructed for more frequent population of the staging table. If such additional load batch identification mechanisms are constructed, the staging table can accommodate multiple loading batches, but at the same time the staging table would be potentially burdened with always rising high watermark or slowed down inserts due to additional indexing requirements.

Assume that a staging table is used to accommodate only the current bucket of incoming data, identified with column bucket_id. The loading process would then load the bucket in a batch job, with any predefined size of a bucket. This process would fill in the table starting from first empty data block of the table extent and continue filling in the extent until the first bucket (bucket_id=1) is handled.

Now, processing the bucket (bucket_id=1) from the staging table further to the data warehousing structures is executed by selecting all rows from the staging table.

```
SELECT [COLUMN LIST] FROM STAGING_TABLE;
```

This will cause a full table scan, as the database engine selects all records in the table. After successfully processing the rows further to the data warehousing structures the staging table needs to be cleansed from the existing data to facilitate for the next bucket_id to be loaded. There are two options to do the operation; either delete all the rows with a delete command

```
DELETE * FROM STAGING_TABLE;
```

or by truncating the staging table

```
TRUNCATE STAGING_TABLE;
```

The delete operation would take considerably long time to execute. This is due to the fact, that the transaction would be logged into the redo logs of the database engine to secure any potential rollback command. The delete operation needs to be explicitly either committed (confirmed) or rolled back (cancelled). With either operation, commit or rollback, the database engine would handle transaction through the redo logs to secure consistency in the data manipulation language (DML) command.

On the contrary, the truncate operation just marks the table as empty and all data blocks in table extent(s) are marked free. The execution of such truncate command is extremely fast as it is irreversible, and the action is not recorded in the redo logs of the database engine. Truncate table is a data definition language (DDL) command. By nature, any DDL commands are not logged into database engine redo logs as they are not processing records, or they are not part of any transaction thus they make the operation execute multitudes of times faster compared to any DML commands.

Now after the deletion of the records in the first bucket (bucket_id=1) or truncation of the staging table, the system will be able to process the next bucket of data (bucket_id=2) into the staging table.

By nature, the traditional method forces the system handle any batches to be loaded in a sequential manner, one at a time, and any parts of the further DWH structure loading process cannot be isolated from the staging table loading process. This creates a heavy dependency between the staging table and data warehousing structure population processes.

Assume that a staging table would be used to accommodate multiple buckets of incoming data, identified with column bucket_id. The population process would then load these in a batch job, with any predefined size of a bucket. This process would fill in the table starting from first empty data block of the table extent and continue filling in the extent until the first bucket (bucket_id=1) has been processed. Let us suppose that the system would process the next bucket (bucket_id=2) to the staging table, e.g. it processes buckets sequentially and not asynchronously, and the first bucket is still waiting to be processed further from the staging table. The processing of the second bucket to the staging table would add the inserted rows at the

end of table; either to the free data blocks in an extent or to a newly added extent and its data blocks.

Processing the first bucket (bucket_id=1) from the staging table further to the data warehousing structures is then executed by selecting rows with bucket_id=1.

```
SELECT [COLUMN LIST] FROM STAGING_TABLE
WHERE BUCKET_ID=1;
```

In this setup, where the staging table allocates multiple buckets of data, the staging table loading process can be by nature asynchronous with the DWH structure loading process(es). Despite the asynchronous capabilities and facilitation for the multiple bucket_ids, this approach has two design flaws;

- Without indexing any SELECT FROM or DELETE FROM data manipulation commands will result into a full table scan
- With additional indexing on BUCKET_ID column, the SELECT FROM and DELETE FROM data manipulation commands will scan through the index, and address only the relevant records of the staging table

Taking the first design flaw example on staging table without any indexing;

Without the indexing any access (SELECT FROM, DELETE FROM) is going to cost additional time and resources as the database engine needs to scan through all data blocks of the staging table to retrieve correct rows for the bucket_id. In this setup the processing times will grow unless the housekeeping of the staging table is executed and timed precisely right to keep the housekeeping and space consumption of the staging table to a minimum level. This is due to allowing multiple buckets per staging table to facilitate asynchronous processing.

In practice this would mean, that the staging table's housekeeping process (e.g. the cleansing of already loaded bucket_id) needs to be executed within the loading process itself after the loading of the bucket_id data and its delivery to the data warehousing structures.

This will introduce a significant delay to the DWH structure loading process and will harm the near real time and asynchronous loading requirements due

to the process needing to delete the records from the staging table at the end of the loading process. The delete operation must execute a full scan to the staging table as there are no indices supporting the addressing of the relevant records. The cleansing process will scan through all the data blocks of the staging table, searching for the right records with that specific bucket_id and deleting those rows. The deletions must be committed in the database. This *DELETE FROM* transaction will write redo logs for the whole operation along with the transaction itself and the commit will make the changes readable for other processes.

```
DELETE FROM STAGING_TABLE
WHERE BUCKET_ID=1;
COMMIT;
```

By introducing the cleansing phase as a mandatory part at the end of the loading process the system would introduce a longer load time per the loadable bucket_id. This would also affect the potential scalability and concurrency on the asynchronous loading setup.

The second design flaw related to staging table with index on bucket_id column comes to additional processing cost for maintaining the index while loading into or deleting from the staging table. This will also introduce additional storage needs for index within the database system. Using an index on bucket_id requires additional resources to maintain the index while loading the data to the staging table. This does enhance the throughput of the selecting the bucket data from the staging table during the loading process but then again, in addition to the redo log generation as in previous design flaw example, this will introduce additional resource needs for index maintenance while deleting the data from the staging table on the cleansing phase.

3.3 The choking effect on near realtime DWH environments

Using a setup with a staging table where multiple bucket_ids would be residing during the load, there are following alternative approaches on:

[1] Populating and managing the staging table without indexing the *bucket_id* column

[2] Populating and managing the staging table with index on the *bucket_id* column

Assume the staging table has no indexing on the bucket_id column and the system is loading multiple bucket_ids to the staging table. Further, the system will not delete the bucket data from the staging table as a part of the loading process but deletes the bucket data as a separate part of the system implementation.

Assume further that loading the data to DWH structures from the staging table may take longer than loading the staging table from the source systems. By implementing the system as described above we can analyze how the system would behave running the asynchronous loading and cleansing processes.

Assume now that the system has loaded five (5) buckets of data, each having 50000 records, thus consisting of 250000 records in total. Then, the record size of 512 bytes (B) would give the space consumption of approximately 125 megabytes (MB) for the five buckets. Adding another bucket would reserve additionally 25 MB of database space for the table while loading the data into the staging table.

Now, loading of the data from the staging table to the DWH structures would be slower than loading the data from source systems to the staging table. Implementing a system like this would force the loading process to halt after some time depending on how much table space has been reserved for the staging table. This is due to the following;

[1] The staging table is a single logical unit of data, built from segments, extents and data blocks within the database
[2] Processing the data from the staging table to the DWH structures is slower than loading the staging table from the source systems (e.g. adding new buckets of data) as assumed above, due to transformation of data within the loading process
[3] The system is not able to delete buckets of data faster than loading new buckets in from source systems due to the slower loading of data from staging table to DWH structures
[4] The table will evidently over time consume growing amounts of space from database due to the fact that loading data from source systems to staging table is faster than data loading from staging table to DWH structures. This yields to longer execution time gradually

on the loading process, as the full table scans will require more I/O resources while the staging table grows.

The above will cause the staging table to consume all free space from the tablespace. The table will grow larger until the tablespace has no more space to allocate for the table and any insertions to the table will generate an error and the system will halt.

3.4 Deleting data from or truncating the staging table

In the traditional DWH method, the staging tables are used for one batch load at a time. E.g. the table holds data for only one bucket at a time. The approach facilitates for easy data management but has very limited support for systems requiring parallelism and concurrency.

By utilizing a fundamentally static staging table for one load batch (i.e. bucket) at a time will secure fast and easy deletion of the staging table data after the data has been loaded. This is eventually executed by a TRUNCATE TABLE DDL command. This method is a fast and robust mechanism to cleanse the loaded data from the staging table and the truncate command is usually executed as the last operation of the loading process that populates the data warehousing structures.

However, the above method allows no concurrency or parallelism in the loading pipeline, e.g. the processing consists of sequential parts executed after each other. Loading the next bucket of data into the staging table requires all parts of this processing pipeline to be finalized successfully. As a result of this, reading of records from source systems to the staging table cannot be parallelized and the processing does not support concurrency.

Approaching the above problem by adding a column to the staging table holding the bucket_id will enhance the concurrency and parallelism. Then, the staging table can hold multiple buckets of data at the same time. It will also facilitate for concurrency and parallelism, as the loading from the source systems to the staging table can be parallelized as well as the processing pipeline that delivers data from the staging table to the DWH structures.

On the other hand, adding the bucket_id column to the staging table will introduce two distinct problems; i) the efficiency aspect and ii) the data skewing aspect.

For the efficiency, the problem is two-fold. Without indexing the bucket_id column, the loading from staging table to data warehousing structures will need a full scan of the staging table for searching the rows with a correct bucket_id. This will make the loading process inefficient, but it introduces the rising high watermark problem described in section 3.1. With indexing the bucket_id, the processing of data from staging table to DWH structures is efficient but the data processing from source systems to the staging table will slow down due to the need of populating the index as the data is loaded into the staging table.

Regardless of the indexing of the bucket_id column the data skewing problem is evident. Skewing in this context means that data is not distributed evenly on specific data blocks but has an uneven distribution across data blocks. The rows of a specific bucket are distributed unevenly across data blocks having empty space. A non-indexed staging table has slower further processing and deletion of data. An indexed table supports faster further processing and deletion of data, but slower data loading into the staging table. The concurrent loading of multiple buckets of data into the staging table will interleave the rows of different buckets among each other while the system is inserting the data into the staging table.

Now, let us assume the system has inserted arbitrary amount of data buckets into the staging table with a parallel processing pipeline. The parallel processing pipeline inserts many buckets into the staging table during a specific time interval. These data buckets are marked as processed and are waiting for deletion. After executing committing the deletion, the physical table extents and data blocks have now free space within them for additional rows. Let us also assume that the database and staging table are generated with such specifications that the database engine can utilize this recently freed space within the data blocks. Now insertion of a new data bucket means that rows are inserted into these partially emptied data blocks and empty data blocks will be reserved at the end of the table. This introduces data skewing to the staging table and introduces growing space consumption for the staging table as loading more data rows to the staging table will reserve additional extents and data blocks for the table.

It is evident and also witnessed through experimenting and resolving the problems in the setup referenced in chapter 5.1, that this setup will ultimately over time introduce excessive data skewing. This will eventually generate a high watermark problem with the staging table. The staging table will then consume additional space from the database until the free space has been consumed. This means that the database engine cannot reserve more space for the table, and any insert operations to the staging table will exit with an error.

Having active DWH system processing near real time data loading mechanism cannot stand such halt. This is due to the fact that the active DWH environment by nature handles data loads in really short intervals. Any delay in the process of loading data from source systems generates a heavier load on DWH loading processes. It is fair to say that another type of solution needs to be formulated to overcome and resolve the problems on both efficiency and data skewing.
All the parts of a typical data warehousing ETL process are affected by the fact that there is a huge amount of records read, transformed and loaded through the system. Each of the parts is affected individually and yet affecting on the total throughput of the system.

Several questions still remain to be answered – for which answers are never good enough. What if there is a need to rerun a day's batch of data? Further, how to rerun in the case of corrupted data? How the correctness of the data can be ensured? How do we process the data fast enough and how do we enable it to be reported on a very frequent, near real-time, basis?

All these questions and design flaws must be answered and refined to achieve a high performing environment for a near real-time DWH system.

4 Existing research

The existing research publications were searched using Google Scholar[1]. Publications were searched in the databases with keywords (data warehousing (DWH), staging table, partitioning) and selected by the author based on their relevance to DWH, their handling of loading processes and staging tables. The publications were reviewed by studying their focus areas and comparing their conclusions and findings to the traditional DWH methods and the proposed AcDWH methodology of the present work.

4.1 1990-1999

Widom [15] studied research problems in DWH. The author described a general DWH architecture and technical issues arising from the architecture. The author discussed wrapper / monitor component that monitors source system changes and provides formatted data to DWH which also informs the integrator component of changes in the source system data. There is a wrapper / monitor component for each data source due to different data models in different source systems. These components also reformat the data to the DWH required data model. The data from the wrapper / monitor components are consumed by the integrator component. The wrapper / monitor components read the data directly from source systems and the integrator component writes it to the DWH. The author discusses how the integrator component will directly write to the DWH structures. The author discussed alternatives with data loading or maintenance of materialized views if a DWH would be refreshed at each query execution. The author discussed the specific extreme case where all data from source systems would be copied into the DWH, and DWH views would be refreshed in entirety from the copied data. The author did not discuss further the

[1] https://scholar.google.com

problems of using data from source systems versus utilizing a staging area within the DWH itself.

4.2 2000-2009

Suresh et al. [21] patented a method and an architecture to automate the optimization of ETL throughput within DWH systems. The inventors proposed a pipelined and componentized approach to ETL workloads where the pipelines are built for different atomic components, each executing specific processing to the data. The transformation server's components separately decide whether to stage or stream the data to be transformed. The pipelines are managed by the transformation server which optimizes the system for maximum resource utilization throughput by parallelizing the pipelines. The user is also able to define parallelism for the system by manually defining how many pipelines the transformation server will handle concurrently. The processing pipelines and their components reside in memory, whereas source data originates from any of the valid source types and the target system is a DWH. The patent describes thoroughly the working principles of the transformation server. The transformation server processes only data in transit through the server.

Bruckner et al. [27] studied approaches to real-time data integration for DWHs. The study discusses an approach that applies continuous near real-time data propagation using integration techniques. The study presented methods available in standard Java 2 platform with a scalable ETL environment implemented with ETLets and Enterprise Java Beans (EJB). ETLets are small ETL components implemented with EJB that execute specific actions, and they have standardized interfaces for the input / output parameters. The authors discuss the business needs of near real-time DWH, namely including continuous data integration, active decision engines and highly available analytical and query setup. They also discuss the differences between ODS and DWHs. ODS is an environment providing view for current state of data across operational systems, DWH is an environment where analytical and historical data are recorded and provided to support business users and analysis. The authors discuss an architecture to streamline data delivery between different layers without using intermediate storage or staging areas. This is achieved by using ETLets and EJB components for extracting, parsing and converting data through J2EE connectors. In their study the authors described how their proposed system

manages the source system connection pooling and ETL processing through containers. By using the container setup on light weight Java components and immediate file storage beneath the authors believe the setup is feasible for near-real time DWH environments.

Nguyen and Min [24] studied a framework of a Zero-Latency DWH (ZLDWH) in 2003. Their article addresses two aspects of ZLDWH; firstly the Continuous Data Integration and secondly Active Decision Engine. The first is constructed from a message queuing system and a data integration tool. This tool receives data from heterogenous sources using a data stream processor and a set of change data detection modules. This part of the system manages the active DWH requirements. The methods can include both push or pull techniques where data are either sent to the receiver or requested by the receiver. In addition, the Continuous Data Integration module can be formulated to handle data either in synchronous or asynchronous fashion, and there can be single or multiple data receivers. This part of the solution uses also continuous data stream processing where data is usually constantly changing, and it is not practical to operate with large data sets multiple times. The second component Active Decision Engine handles the rules and actions within the system. Its primary function is the automation of different tasks by analysis rules which are created traditionally by incremental analysis of the collected data. The Active Decision Engine uses a rule base, an event base and an action base to handle the automation. Users are able to create and modify different rules, events and actions through a specific end user interface or tool. There are some foundational problems in applying continuous data stream processing related to time consistency. These problems are evidently introduced by the process, which realizes when the data has been valid so that the data can be processed with properly modelled dimension data including attributes for validity time and data load timestamp details.

Golfarelli et al. [16] discussed the horizon of beyond DWH in terms of looking onto what will be the next trends in business Intelligence. The authors discussed the data freshness needs related to decision making for an organization, while trying to execute the company's strategy. The paper discusses different aspects and needs for data and information, indicating that Business Performance Management (BPM) is a potential resolution for data freshness for decision making of an organization. The research indicates that the data needs to be continuously made available at the right time and in the proper format to the right decision level makers. Decisions on lower organizational levels require more fresh data due to the decisions

need to be made faster. The above referenced BPM systems provide data and insight in the right time, instead of real-time, facilitating fresh enough data. In addition to the freshness of the data, the lifetime of data is relatively short. The data are needed for the dashboard usage on current performance metrics. This can be achieved in reactive data flows, that monitor the processes with time critical aspects. This kind of activity is called Business Activity Monitoring (BAM). The main components of such a construct are Right-Time Integrator (RTI) and Dynamic Data Store (DDS). RTI is an engine integrating data from operational databases, DWHs, Enterprice Application Integration (EAI) systems and from real-time streams. DDS system is storing short-term data for fast retrieval needs and mining. As data latency is of key relevance, the article proposes abandoning the Operational Data Store (ODS) approach utilized in the DWH and concentrating on on-the-fly techniques. These techniques utilize BAM approach, implementing right-time processing of the relevant data. As a conclusion, the BPM approach and its role is seen as a method to quantify the strategy and targets and to facilitate decentralized decision making on the operational and tactical levels of organizations.

Karaksidis et al. [20] studied utilizing ETL queues for active DWH for maximum freshness of the data. The authors discussed different approaches to build an active DWH, such as data streams compared to traditional method of loading windows during night and offline population of the DWH. The authors divided their study around four main requirements: maximum freshness of data, easy and swift upgrade of software at the source systems, minimized overhead to the source system and stable interfaces at the DWH side. The study discussed an active data staging area (ADSA), from where the data are loaded into the DWH utilizing on-line loaders. The authors employed a queue for each ETL activity, namely building an ETL data flow of separate queues processing data in different manners; e.g. the processes are different consumers of data. The system architecture consists of a data store (DBMS, application or similar), source flow regulator for handling the data flow from data store, intermediate staging area (ADVA) where data are cleansed and transformed, web services for consuming data from ADVA and populating the data into DWH. The authors proposed alternatives for the staging area, first being on the source side, second on the target DWH side and third one as a separate environment. The study specifically addressed the choices concerning staging area. According to the authors, the internal staging area structure and its tuning are the key elements of the architecture and its performance. As the staging area is an environment of multithreaded nature and it is using

shared coonstructs, race conditions and consistency should be handled properly. The authors raise issues on the locking of the queues and its implications to how fast the queues can be handled and emptied. Too fast arrival rate of data from source systems generates instability and longer queues. Alternatively, too fast service rate transmitting data off the queues will create a lot of locking issues, thus arrival and service rates should be close to each other to avoid problems. Also handling the data one tuple at a time poses a large overhead to the system compared to an approach where data are handled one block at a time. As a conclusion the study summarized the findings as follows; the proposed system with isolated ETL tasks to a specific area adds very limited additional costs to source side and the proposed system also facilitates faster flow towards the DWH.

Simitsis et al. [10] addressed the optimization of the ETL processes. The authors reviewed and focused their approach to logical transformations of the workflow instead of implementation requirements on the physical side. The study approaches the optimization problem of ETL processes through different algorithms, and their effect on the outcome. The optimization techniques included exhaustive and heuristic techniques to ETL workflows. The study gives a comparison of the different approaches and their impact to the execution efficiency. The study concentrates fully on the logical ETL workflow and does not address any physical side design elements. The authors disclosed that the research issue of physical optimization of ETL workflows has been left unexplored.

Polyzotis et al. [23] researched utilizing streaming updates in an active DWH. The authors studied particularly an active DWH research problem where transactions are inputed through online data streams. Transactions are added with details from a DWH table such as a dimension table, where the transaction is added with surrogate keys looked up from dimension tables. When using the traditional ETL lookup setup, the ETL logic reads the full lookup table in the cache memory for the specific invoking of the ETL process constituting from multiple rows. For streaming data sets the lookup caching problem generates extensive overhead due to nature of the processing a record at a time. For this problem the authors proposed to use a configurable mesh join, which keeps a specified amount of mesh join attributes in the memory to avoid re-reading of the lookup table. The researched mesh join can be configured to either stay within a specific memory limitation or to handle the incoming data stream at incoming rate. The algorithm skips processing any results that are already in the lookup table in the memory and propagates new results to the in-memory result

table. This result table is kept in the memory instead of looking it up from the database for each execution of the ETL process. The construct amortizes the cost of reading the lookup table over a set of tuples and thus provides far better efficiency than typical ETL processing but at the same time it consumes more memory.

Santos and Bernardino [9] proposed a continuous loading mechanism for real-time DWH. The method adapts the DWH schema by duplicating the DWH tables into temporary tables which are identical to the original DWH table and added with a unique sequence identifier column. These temporary tables are created without indices, primary keys or any constraints. To refresh the DWH, all new data are loaded into these temporary tables with autoincremented sequential identifiers. Any queries to the DWH will be adjusted to query from both temporary and actual DWH tables. This method will in time cause the slow-down of the insert and query operations. The system can be optimized by moving all newly inserted data from the temporary tables into the DWH tables.

Polyzotis et al. [25] studied meshing streaming updates that use persistent data in an active DWH. The authors studied the drawbacks of traditional DWH data loading on nightly basis. The study discussed a specific join of fast source system stream originating updates (e.g. fast paced changes) to a disk-based relation (e.g. a database table or similar). The authors proposed a mesh-join algorithm where the algorithm keeps the lookup table on the disk assuming the available memory is not large enough. The proposed mesh join solution keeps a disk-based relation continuously open, performs a cyclic scan of it continuously and maps the records against the stream originating records. The study focused on the transformations (lookups, joins and similar) in the ETL process.

Naeem et al. [29] studied an event-driven near real-time data integration architecture. They presented an architecture for an event driven near real-time ETL layer using database queues (DBQ) which is working with the push principle. The study describes the foundational problem of continuous extraction and transformation of data within a limited loading window. This problem occurs especially in the management of so-called master data, which is needed to enrichen the transactional data originating from the source system(s). The authors gave a method how the master data can be utilized in an efficient manner through storing it in a separate repository. The master data and transaction data are distributed to right targets and repositories. The transaction data is then enriched with the master data

through a message driven bean which uses the master data tables as inner tables in a join loop providing efficient throughput. Using this method, the master data is not needed to be refreshed for each transaction but rather as the master data itself is changed.

Seifert [5] filed for a patent on an online table move method. The author developed a method to move a table in an online fashion without interruptions to applications using the database. The basic principle is to initiate a module that records all source table operations to the target table, to establish a copy of the existing table, and to initiate replay, swap and cleanup modules. The method uses a staging table to record any changes in the source table while the data is copied to the target table. The swap module will implement the name change of the source and target tables, so that the target table will be established as the table in use. After data has been copied from the staging table to the target table, the cleanup module will delete the staging and source tables. The staging table is not partitioned but indexed for the access of the changed records. The access of the staging table relies on the indexes, and data are processed by reading the entire staging table without parallel processing.

Jörg and Deßloch [8] studied near real-time DWH using state of the art ETL tools. According to their study the requirements could be fulfilled using traditional ETL tools and by shortening the DWH loading cycles. This would not require re-implementation of any of the transformation logic. The study is divided into sections, discussing the refreshment anomalies, concepts of incremental loading and properties of operational sources. The refreshment anomalies happen when DWH system addresses the source systems' data and their changes during the refreshment cycle of a DWH. Two families of algorithms, eager compensating and strobe family, were discussed and their potential to be constructed using ETL tools. Both, the eager compensating algorithm and the strobe family algorithms are tracking changes in source systems while data warehouse loading is executed. Respectively they perform specific compensations for avoiding anomalies. The authors came to a conclusion where the current ETL tools do not provide a means of implementation for the algorithms as such. Incremental loading aspects were discussed in detail, giving simple examples on different approaches. The authors considered the options of full and incremental reloading, and the distinctive characteristics that both approaches introduce. Operational sources and their properties were presented, having a view for example on snapshot and logged sources. Snapshot sources are simply operational sources that allow their material to

be dumped periodically into a file system representing a state of the operational system at the specified extraction time. In this setup change data can be captured using successive snapshots and by comparing states between the different snapshots. Some operational sources implement a change log that can be utilized to extract the changes. There are multiple possibilities to implement change data capture, like triggers included into transaction logic or log-based change capture recording the changes for example to log tables. In addition, database log scraping or sniffing implementations are discussed, in which the source system changes are collected from the active database log files instead of recording the changes from source databases themselves. Often the source systems contain timestamped source data, where the changes are recorded into the source system data itself, an example being the timestamp of the record being created or updated. The authors conclude with showing the potential of using low latency updates using ETL tools in a micro batch manager setup, where the loading cycles and amount of data are strictly limited. This is to avoid refreshment anomalies and subsequent inconsistency in DWH, implemented with different techniques in the ETL workflow and change data capture setup.

Chakraborty and Singh [31] studied a partition-based approach supporting active DWH streaming updates. The authors described the same problematics as Polyzotis et al. [23]. Based on the observations in the study, they proposed an approach to join a data stream with a persistent relation, e.g. a lookup or dimension table using partitioning. The dimension table is divided into partitions, where the join relation can be limited to a limited amount of partitions from the dimension table and potentially the amount of partitions kept in the memory will be adjusted. Additionally, the proposed solution also addresses the I/O bottlenecks and eliminates locking factors which are due to writing rows into the dimension table. This is achieved by maintaining a wait buffer which is not written to disk, but rather kept in memory. Compared to [23], Chakraborty and Singh have added a distinctive partitioning method on the top of the proposed mesh join setup, which efficiently eliminates large scale reading of the dimension table and tries to concentrate the reads and writes into hot areas within the partition range.

Vassiliadis and Simitsis [12] discussed the business needs that require near real-time DWH and such architectures that will cater for these needs. The authors considered also the performance bottlenecks relating to near real-time data loading, especially arranging the data into a staging area, and processing it into DWH or data mart structures either using bulk loading

mechanisms or inserting the data by a sequential insertion of rows. Vassiliadis and Simitsis properly identified the drawbacks of such mechanisms in relation to indexing and materialized views over the DWH relations while inserting the data. The authors proposed an Extract-Load-Transform (ELT) solution that snapshots operational system data into DWH staging area, after which the transformations are managed within the DWH platform. This enhances scalability and also secures integrity of data as all data are kept within the database engine. The proposed solution is a pipelined approach to the stages in the ELT process where a proposed Data Processing Flow Regulator (DPFlowR) component controls the source loading activities and decides which sources are ready for transmitting data. This proposed component also regulates the source loading process and balances the congestion posed by the loading pipeline against the overall system throughput and responsiveness. In addition, a proposed Warehouse Flow Regulator (WFlowR) component would similarly control the DWH loading processes that are pipelined, balancing them to enhance system throughput and responsiveness. Both these components act as load-balancing tasks within the system. The authors discussed specific pipelining and partitioning methods for the specific extraction, loading and transformation processes. The discussed method proposes to divide the processed data into smaller sets which would then be processed in parallel and in pipelined fashion by different parts of the system.

4.3 2010-2019

Zuters [13] studied near real-time DWH problems and proposed a solution to data loading setup by evolving of trickle & flip method into a multi-stage trickle & flip setup. This trickle & flip method is used to remove scalability issues in DWH for querying the data which has been updated concurrently with the querying processes. Using trickle & flip the staging tables are in the same format as the DWH tables. The staging tables are periodically duplicated, and their copy will be swapped with the DWH tables. Applying trickle & flip to real-time DWH means swapping the staging tables with the active partitions of the DWH. This method implicates that the system needs to have all changed data available since the last update in the real-time portion of the DWH. In addition, the real-time data needs to be linked pragmatically to static data, it needs to be extremely lightly indexed to support the continuous data loads and to support fast queries. Using trickle & flip imposes drawbacks to the real-time DWH setup. Copying staging

data into DWH active area for example every hour will implicate periodic slowness on throughput and tweaking the update happening in longer intervals just exaggerates the impact as the data swapping will take longer time. Zuters proposed using an evolved multi-stage trickle & flip scenario where the method introduces additional stages to the system. This would resolve the issues of querying the tables while the data are loaded. In this scenario the real-time data are divided into sub-partitions, where each sub-partition holds less data than the full real-time portion of the DWH. The staging table is proposed to be swapped in more frequently, and then only a sub-partition of the data must be moved. This efficiently removes some of the hindrances of querying and loading the same data window.

Thomsen and Pedersen [11] presented an ETL framework implemented in python programming language. The framework presents an efficient way to parallelize the ETL process itself and the typical tasks of such process. The research addresses several constructs in python that will enable all parts of the ETL process to be parallelized. This is achieved by both task parallelism and data parallelism. The proposed method allows extraction in parallel to other tasks in the process. Authors proposed to divide the tasks into flows that are sequence collections of functions running in parallel. The method enables the programmers to decide and control which parts of the ETL process and data can be parallelized.

Kakish and Kraft [26] studied the ETL evolution for the real-time DWH. The authors presented the fundamentals of the ETL processing in traditional DWH environments and described the architecture of the DWH environments. The authors discussed the problematics of capturing changed data from the source systems and the complexity of defining extraction processes. Kakish and Kraft described the techniques to achieve real-time DWH through implementing a Change Data Capture (CDC) technique and integrating such technique with ETL tooling. In CDC technique only the source system changes made after previous extraction are extracted. So, CDC mechanism uses only incremental extraction. This integrated approach would minimize the needs for resources along with maximizing the efficiency of the process. The study describes the three different generations of ETL toolsets, which have evolved from operating system native code, through proprietary ETL engines to latest generation of ETL tools which have a distributed architecture. These third generation ETL tools eliminate and reduce the need for an ETL hub between the systems and they pursue to introduce distributed processing where the transformations are implemented in the database management system side. This facilitates for distributed and

optimized ETL processing. The authors came to a conclusion where the current ETL processes need to transform from periodic processing to continuous updates. According to the authors, effectively this would require continuous data integration. To eliminate the disadvantages and to fulfill the requirements, authors propose to use an intermediate data processing area (DPA) and the architecture and methods proposed in [12]. The study concludes with weighting the different aspects of different solutions on the actual need; not all tasks require real-time analysis capabilities.

Waas et al. [6] discussed the problematics of near real-time DWH in the context of the latency to get the data in the DWH for queries. The data freshness problems were discussed. A core problem of data freshness and latency related to time consuming data transformations and cleansing for queries was identified. The authors propose a right-time Business Intelligence (BI) architecture where ETL is turned into ELT processing using database platform as the loading and transformation engine. The paper proposes loading raw data into the DWH and handling the rest of the ELT process with database operations through materialized views. The proposed model has three main components: staging area called landing pad (LP), DWH tables, and materialized view stack (MVS) providing data to the queries and reporting instead of traditional data marts or reporting tables. The data is provided to the queries on-demand through refreshing of materialized views. Authors also proposed to augment the architecture with updates through streaming data from event data sources. The streaming data process can query and combine elements from DWH for end user dashboards for alerting.

Bani et al. [35] studied utilizing Massively Parallel Processing (MPP) system to provide scalability for DWH. The study focused on implementing a MPP system with Greenplum database to perform complex queries in the DWH. The Greenplum MPP system is built with multiple parallel physical hosts interconnected with an interconnect network layer distributing the MPP processing. The DWH data is partitioned across the servers and each server has its own CPU, memory and database instance. The database queries run in parallel using all the MPP system hosts, and each host is returning the results. Interconnect network layer enables communication across the database instances residing on the servers, giving the system ability to act as a single database. The MPP solution collects daily transactional data from the source systems. The study shows that data loads with less than 1 000 000 rows can be handled with direct load to staging area tables, and the staging cleansing is executed by truncating the staging

table after each successful load to DWH. Larger data loads require a dual stage data load which means creating a file dump on source system table(s) and loading the data from these files to staging area using databases utilities.

4.4 2020-

Gorhe [36] studied problems and categorized challenges and opportunities in ETL processing for near real-time environments. The author identified fast source data availability in DWH environment and providing required data for decision making as the primary focus in near-real time DWH. Low latency, minimum disruptions and high availability & scalability were identified as the key characteristics of these near-real time DWH environments. The author also discussed problems in the ETL processes. Some key findings were performance impact of the DWH while loading the data, the inability of proprietary ETL toolset to support near-real time usage and complicated design due to the near-real time requirements. The author identified key findings on the opportunities in near-realtime DWH, such as data buffering to enable source data storage while previously extracted data was under processing and using separate ETL for near-real time data.

Adnyana and Jendra Sulastra [37] studied data backup and synchronization implementation for real-time DWH. The authors considered the resolution of data synchronization to online transaction processing (OLTP) systems and DWH databases while network problems occurred. They described a functionality in the system which saves the data into a comma separated values (CSV) file while network problems occur. The solution uses an identity column on OLTP database tables to mark if the insertion has failed or succeeded. After the insertion has succeeded to the OLTP database, the system continues synchronizing the data into the DWH.

Biswas et al. [38] studied incremental loading techniques for real-time data integration. The authors compared Graphical User Interface (GUI) -based ETL tools in the market against custom coded tools. The study discusses four programmable ETL tools Pygrametl, Petl, Scriptella and R_etl. The authors described and measured the efficiency of each of the programmable ETL tools from different viewpoints and discussed the modelling of the ETL jobs. The authors divided their study to three parts: Change Data Capture (CDC), dimension table processing and fact table processing. The authors experimented with full reload against incremental load and they

came to a conclusion that incremental loading is not only faster but also provides lighter processing requirements for the system. Conclusions include findings that specifically coded and crafted ETL can be the most viable option instead of GUI-based ETL tools on the market, and also that real-time DWH needs incremental loading mechanisms which provide better throughput and also less system resource consumption.

Cao et al. [39] presented Timon, a time-series database implementation for efficient telemetry data processing and analytics. The authors created the solution for timestamped event database that supports aggregation and handles late arrivals. Timon uses TS-LSM-Tree structure that keeps recent data within memory. The structure also contains a time partitioned tree on disks to which the in-memory data is periodically merged to. The non-memory implementation is usually done with such solutions as HBase [40] or Cassandra [41]. Timon reads the events from the source systems usually through message queue systems and attaches a sequential ascending identifier to each record. The solution is built to support large volumes of timestamped data. Timon is written from scratch, and the authors have implemented also Timon Query Language (TQL) for easier application development.

4.5 Summary of literature review

As a summary, below is a comparison of different focus areas in the prior literature reflecting the area of this thesis. While most of the cited studies focus on the loading process and the staging area handling only a few of them focus on table partitioning setup and associated methods to overcome active DWH bottlenecks [23] [31] [39]. While the table partitioning is studied in these papers, it is not studied for the staging area handling. Table partitioning aspects have been studied in the context of loading process or join processing, which are elementarily valid focus areas. The present thesis uses table partitioning in the staging area processing. Also leveraging the partitioning technology to enable active DWH with continuous loading and simultaneous querying of data is be studied. As a conclusion, utilizing table partitioning in staging tables in a standard database engine has not widely been discussed or studied. This thesis proposes a novel approach to active DWH staging table handling in a standard database engine using table partitioning along with the proposed data management system.

Table 1. Comparison of focus areas in prior studies.

Referenced study	Author(s)	Year	Loading process	Staging area	Join processing	Table partitioning
[15]	Widom	1995	X			
[21]	Suresh et al.	2001	X			
[27]	Bruckner et al.	2002	X		X	
[24]	Nguyen and Min	2003	X			
[16]	Golfarelli et al.	2004	X	(X)		
[20]	Karaksidis et al.	2005	X	X		
[10]	Simitsis et al.	2005	X			
[23]	Polyzotis et al.	2007			X	X
[9]	Santos and Bernardino	2008	X	X		
[25]	Polyzotis et al.	2008			X	
[29]	Naeem et al.	2008	X			
[5]	Seifert	2009		X		
[8]	Jörg and Deßloch	2009	X			
[31]	Chakraborty and Singh	2009			X	X
[12]	Vassiliadis and Simitsis	2009	X	X		
[13]	Zuters	2011	X	X		
[11]	Thomsen and Pedersen	2011	X			
[26]	Kakish and Kraft	2012	X	X		
[6]	Waas et al.	2013	X	X		
[35]	Bani et al.	2018	X		X	
[36]	Gorhe	2020	X	X		
[37]	Adnyana and Jendra Sulastra	2020	X			
[38]	Biswas et al.	2020	X			
[39]	Cao et al.	2020	X	X		X

5 AcDWH Method

In this chapter a methodology for a rapid data warehouse (DWH) loading and analysis platform (AcDWH) is presented. The methods presented have been granted a European Patent (EP 1 959 359 B1) by European Patent Office on November 22nd, 2017 [I].

5.1 Overview

A high-level description of the optimized DWH loading and analysis platform, AcDWH, is as follows:

1. Generate the AcDWH staging area and primary data warehousing structures to enable initial loading,
2. Feed the AcDWH staging area from the source systems (data loading module),
3. Populate the primary AcDWH structures from the staging area (delivery module(s)),
4. Clear the staging area after transitioning the data to primary AcDWH structures (cleaning module),
5. Establish the AcDWH indexing structures for analytical and query use.

The foundation of the AcDWH methodology is explained in detail in the following sections. The fundamental change of the proposed approach in contrast to the previous DWH techniques is utilizing physical data partitioning in a manner it was not originally intended to be used.

High level modules and their relation to the staging table(s) of the AcDWH are illustrated in Figure 4. Step 2 is performed with the loading module, step 3 with the delivery module and step 4 with the cleaning module. Step 1 is

performed manually while building the system and step 5 can be performed by triggering indexing structures recreation as a last part of delivery module. Each target table / structure has its own system modules. Each bucket is represented by a single table partition.

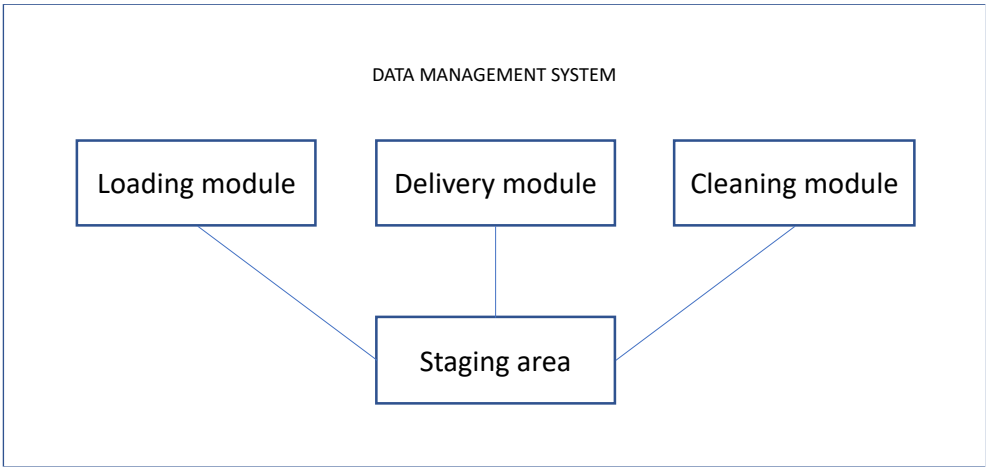

Figure 4. AcDWH system modules and their relation to the staging table(s).

High level flow of the AcDWH load module is shown in Fig. 5:

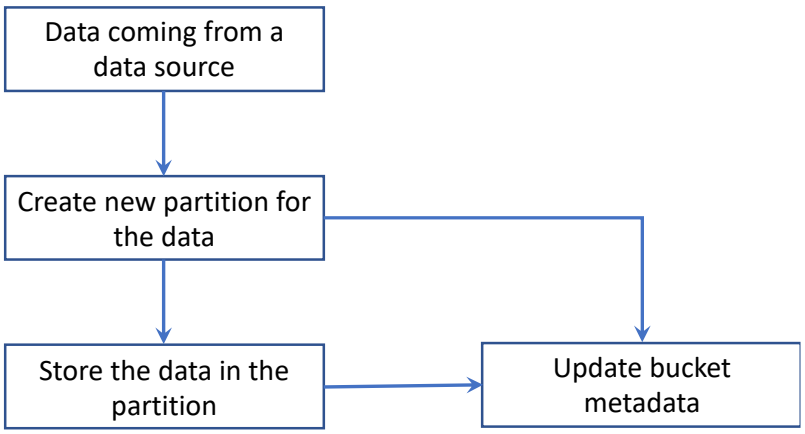

Figure 5. High level AcDWH loading module.

Each bucket in the subject area specific staging table is created with uniform extent and bucket sizes, meaning physical table partition size and also as

close to an uniform amount of rows in the bucket as possible. After the bucket has been loaded into the AcDWH staging table partition, it will be marked as loaded and any delivery processes can start to load data from the staging table to DWH structures.

The AcDWH system may include multiple delivery processes and thus the system needs to track how the different delivery processes will load data into database structures. Multiple delivery processes can load same bucket data for example to different subject areas (data marts, DM) for reporting. The data bucket from staging table cannot be deleted prior to all delivery processes have processed it. To facilitate for this the system works through the control tables and coordinates how different delivery processes work with the data. See Figure 6 for the workflow of the AcDWH delivery module.

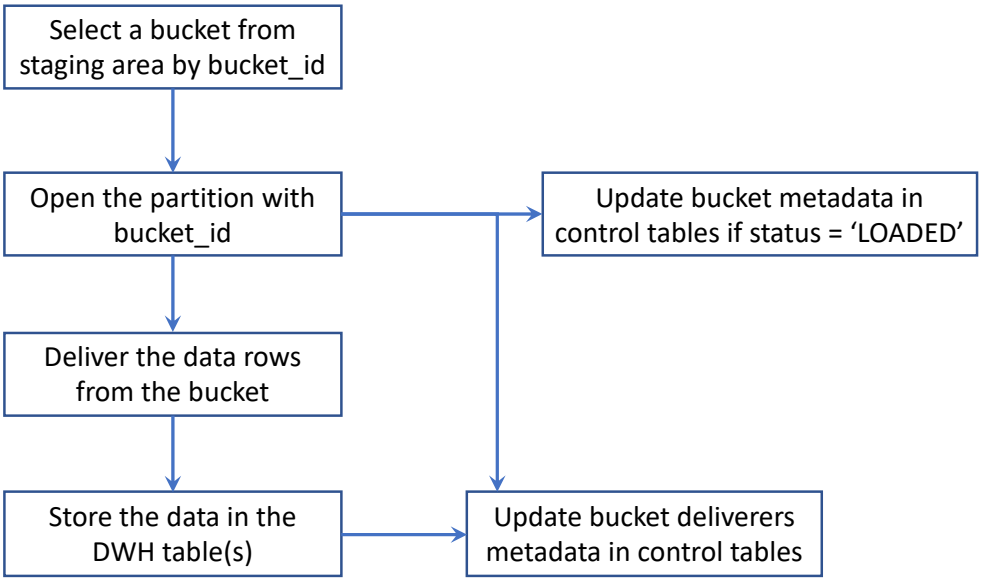

Figure 6. High level AcDWH data delivery module.

To set different priorities on the delivery processes, each AcDWH delivery process is assigned with priority information. The priority is indicated with a numerical value. Value 1 has highest priority, value 2 second highest and so forth, as many priority levels as needed can be introduced. Priorities of delivery processes are defined for the application while the system is built. The AcDWH system can resolve prioritization by checking if higher priority delivery processes are in the queue to be executed. This simple method

avoids executing lower priory delivery processes before the higher priority ones. When higher priority processes are finalized, the lower priority processes are processed. The overall AcDWH delivery process is described below in Figure 7.

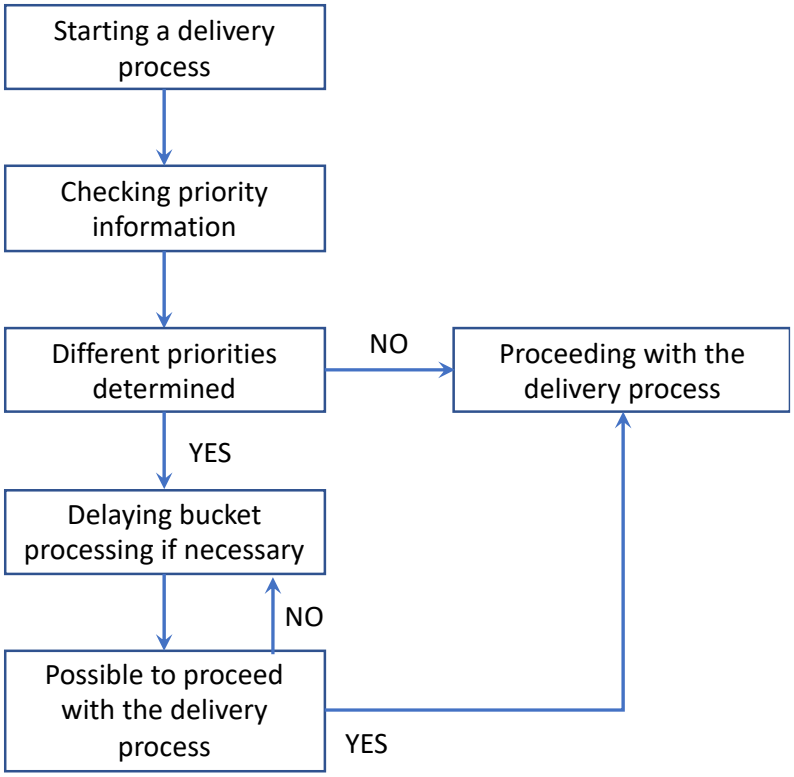

Figure 7. AcDWH Data delivery module priority determination.

After all delivery processes have moved the bucket data to AcDWH database structures, the data buckets (e.g. staging table partitions) will be deleted from the system to minimize the usage of the database space and the consumption of resources. As the AcDWH staging table and its partitions use uniform sizing on physical level, the space removals and allocations are uniform, and the space allocation management is easy.

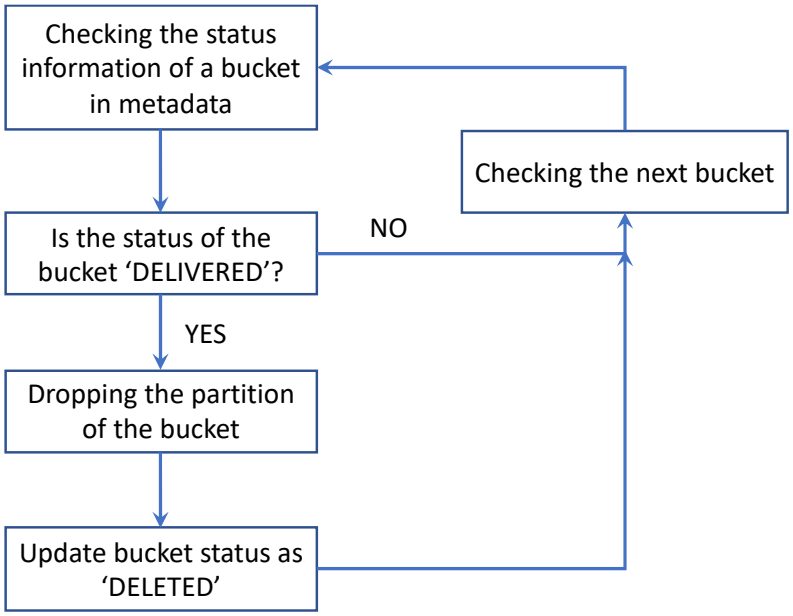

Figure 8. AcDWH Cleansing process.

Figure 8 describes the overall AcDWH process for cleansing of the partitioned staging table from the processed buckets. The cleansing module identifies the delivered buckets from the AcDWH control table (see chapter 5.4), and one by one drops them from the partitioned staging table (see chapter 5.3). This is an efficient way to purge the already processed data from the staging table.

The cleansing process can be established in a very simple manner. The process analyzes from the control table if all the delivery processes have loaded the bucket data from the staging table to the structures and deletes the associated staging table partition. This will be repeated for any potential additional staging table partition until no such partitions are found.

The cleansing process can be scheduled to be run periodically depending on the need. Heavily loaded systems require more frequent cleansing of the staging table and thereby the cleansing process might be scheduled to be run for example every five minutes. In lighter loaded systems cleaning might be scheduled to one hour's schedules or even longer, for example once per day.

5.2 AcDWH structural considerations

The *traditional method* of loading data in large nightly batches into DWH structures introduces performance problems on simultaneous queries and analysis. The loading of data requires typically indices to be put offline or dropped and rebuilt or recreated after the loading of the data. This causes the system to perform a full table scan of the tables when making queries or analysis on the DWH structures. Querying multiple tables joined together without indices will render the system nonresponsive and unusable.

While the indices are turned off the loading of data to the staging area is fast. On the other hand, any query issuing a full table scan slows down the loading of data. This is due to the database engine scans the same physical extents of the staging area as the loading of data process. This is the reason why the data loading is commonly processed during night on daily, weekly or monthly intervals.

The traditional method has also another drawback. As the indices will be put offline or dropped during the data load, the indices need to be either rebuilt and made online or recreated. Taking into consideration that data warehousing tables typically are large and include millions, and sometime billions, of data rows the rebuilding or recreation of indices will be extremely time and resource consuming. This will cause additional resource problems and delays on getting the data ready for queries and analysis.

The proposed AcDWH method will help overcome the above problems by partitioning the DWH tables into smaller partitions having local indices. A local index means that the index will be partitioned according to the partitioned table. Now loading data into a partitioned DWH table will address only the specific partitions it needs to insert data to, and the indices can be kept online while loading data as the partitions are separately addressed. This is a near perfect way to manage large data warehousing tables and it allows simultaneous data loading and querying access.

5.3 Generating the AcDWH structures

The new methodology used in AcDWH consists of the following atomic elements on generating the structures:

- Staging table structures
- Primary DWH structures
- Indexing structures

The AcDWH methodology relies heavily on the special organization of the staging table structures. In addition, by rearranging the staging table, the system is able to provide a widely parallelizable process for querying and loading the DWH.

Utilizing traditional method on generating staging table structures is simple and straight forward. The staging table(s) are generated according to the source system specifications. The staging table has the same columns and data types as in the source system definitions.

The AcDWH method discloses a staging table structure to manage and handle vast amounts of data from source systems, with the ability to handle the problematic areas of efficiency, repeatability and concurrent read/write access to primary DWH structures.

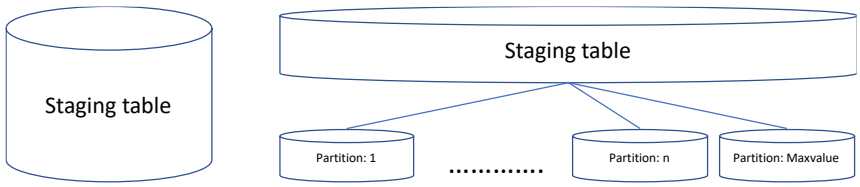

Figure 9. **To the left, traditional staging table and to the right, the partitioned staging table used in AcDWH using uniform partition physical size**.

The novelty with the proposed AcDWH method grants a concurrent and efficient read/write access to the staging table, while the staging table can be simultaneously written into, read from, and cleansed from data originating from several different source systems.

The primary idea in the implementation of the AcDWH staging tables is to partition the staging table into physical partitions. By this method, one can limit the write and read accesses to dedicated physical objects within the staging table. This method is illustrated in Figure 9.

The staging table has additional two columns to source data, namely bucket_data_type and bucket_id. Bucket_data_type, as defined in the

system meta data, defines the source system from which the data arrives. The bucket_id has an identifier for the bucket, e.g. it is a load batch identifier.

The staging table is initially created only with one physical partition, having the partitioning key defined as MAXVALUE. MAXVALUE is a specific value, which does not correspond to any created actual physical values within the bucket_id.

As the loading process gets a specified number of files or rows from source systems, it will generate a new bucket_id. The loading system generates and/or updates a row in bucket_information status table to manage the status of the incoming data buckets. The loading system also generates a new partition in the staging table having the bucket_id as the partitioning key. The new partition is generated from the MAXVALUE partition by a split partition command. This is illustrated in Figure 10.

This way the system will split the MAXVALUE partition into two physical partitions; bucket_id (first load batch being number 1) and MAXVALUE. In AcDWH the split partition command is always executed against the MAXVALUE partition, which is empty, thus no data movement is required and the database management system is not required to transition any rows between the MAXVALUE and newly generated partitions.

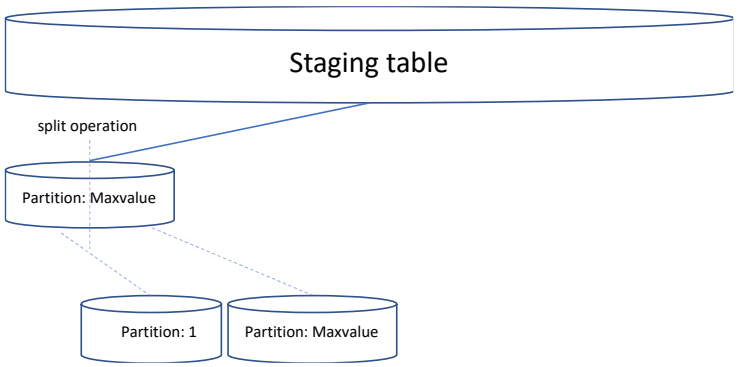

Figure 10. Illustrating the AcDWH split partition operation where MAXVALUE partition is split into partition 1 and partition MAXVALUE. All partitions use uniform physical size.

The methodology described above directs all insert/update/delete database operations into a specific physical table partition. The rows within each

physical partition, having a specific bucket_id, will be directly accessible by the database management system for the loading mechanism without additional or user generated indexing for the bucket_id. In this method the staging table is managed by physically partitioning it by bucket_id.

This method will relax the system from the requirement of indexing the bucket_id column. By selecting rows by bucket_id from a staging table built in this way will always direct the database engine to select the rows from the wanted physical partition only. Such selects do not need additional indexing, and the select operation will also be executed faster and also in parallel by the database engine.

5.4 Parallel processing in AcDWH within a single bucket_type and between different bucket_types

The proposed AcDWH method gives clear advantage over the previous way of populating the staging area and DWH tables. As each data bucket is placed within a specific staging table and a specific physical partition of the staging table, the system is able to insert multiple buckets at a time to the same staging table regardless of the progress on other streams and processes populating the same staging table. Concurrent streams of a particular bucket type are not dependent of other types, or they don't race for the same resources within the database.

Using the AcDWH method, the system is able to load and query concurrently the specific staging table with a greater number of processes with a minimal impact on resource race and consumption. While a load process (from source systems to staging table) is not finished with loading the bucket into staging table partition, no other loading or delivery processes will access the partition. After the bucket is loaded into staging table (and its partition), loading process(es) are allowed to access the bucket. Simultaneous reading of the bucket data (staging table partition) is allowed and does not create race conditions. After all delivery modules reading the bucket data from staging table partition are finished, the cleansing module is allowed to access the bucket and remove its data and its underlying table partition.

This is achieved through physical isolation of the underlying partitions of the specific table. Each table partition is formed of specific physical segments of data placed within the database engine. During the loading of rows into the staging table, the database engine directs the insert operations into the specific table partition, based on the partitioning key (bucket_id) of the table. A given partition is formed of one or many physically separated partition extents. Any insert operation having a specific bucket_id will be directed to a specific physical extent of a partition, and that extent is physically separated from other partitions of the table. The same will apply for the select operations, when partition data is queried to offload the data to actual data warehousing tables for reporting and analysis.

The partitions of the staging table are formed from physical extents which are defined during the creation of the table and partitions. Each partition can have a partition specific extent space. By measuring the responsiveness of the system on different configurations, along with experience on creating and managing the system referenced in chapter 6.1, the system can be configured to allocate only necessary number of physical extents for a specific partition when the partition is created. Using minimum amount of extents for a partition the database engine does not need to allocate any time or space for the bookkeeping of extents within a partition. This will influence the system throughput as the system does not need to automatically allocate additional extents as the previous extent is filled with data. Additionally, the database system does not need to search the starting address of the next partition extent within the tablespaces and database data files. Tablespaces should be always created large enough to hold additional data and the AcDWH system will monitor the space consumption and alert the database administrators should the free space fall below a predefined threshold, such as one day's data space requirement.

The method of dedicating a physical extent within a partition segment for a bucket_id is the foundational element for the achieved concurrency and effectiveness. The system can run a high level of concurrent insertions of bucket_ids to the staging table while at the same time the direct addressing capability for the bucket_id is maintained.

The status of each bucket is kept up to date within a control table by maintaining meta data for each bucket:

- status of the bucket (processing, processed, deleted)
- number of rows
- bucket_type

- earliest record of the bucket
- latest record of the bucket
- start and end times of bucket load
- calculated throughput (as rows processed per second)

The table row size is depending on bucket_type, some staging tables might have fixed row length while others might have variable row length due to variable length data.

Figure 11 illustrates the three status tables that control the behavior of data management system.

CTRL_BUCKET	
BUCKET_ID	NUMBER(10)
BUCKET_TYPE	VARCHAR2(20)
NUMBER_OF_ROWS	NUMBER(6)
EARLIEST_RECORD	DATETIME
LATEST_RECORD	DATETIME
STATUS	VARCHAR2(20)
BUCKET_STARTTIME	DATETIME
BUCKET_ENDTIME	DATETIME
ROWS_PER_SEC	NUMBER(6)

CTRL_DELIVERER	
DELIVERER_ID	NUMBER(4)
DELIVERER_NAME	VARCHAR2(20)
DELIVERER_TYPE	VARCHAR2(10)
DELIVERER_DESCRIPTION	VARCHAR2(100)
PRIORITY	NUMBER(2)
DELIVERER_TARGET	VARCHAR2(20)

CTRL_BUCKET_DELIVERERS	
BUCKET_ID	NUMBER(10)
DELIVERER_ID	NUMBER(4)
DELIVERER_STARTTIME	DATETIME
DELIVERER_ENDTIME	DATETIME
ROWS_PER_SEC	NUMBER(6)

Figure 11. The control tables used to control the AcDWH.

The control tables include data for each loaded bucket in the system. The main control table CTRL_BUCKET is used to record the buckets into the system, in addition the table has metadata related to the loading process efficiency, loaded data time span and bucket status. The second control table CTRL_DELIVERER is used to describe the delivery processes, their types and target structures, and their priority (as described earlier in this chapter 5). By using priorization, AcDWH system can set different priorities based on the different needs of delivery processes. The last control table CTRL_BUCKET_DELIVERERS is used to log the activities of each delivery process with regards to the specific bucket, it can also be used to execute the cleaning module when all the delivery processes have transferred the data to the needed database structures.

As the bucket_id is the partitioning key for the staging table, the direct access path to physical table partition is always up to date and available for inserts and queries.

A typical approach in DWH solutions having large data volumes is to make all indices offline while inserting data and rebuilding the indices online after the batch data load. Another approach is to drop the indices prior to batch loading data and recreate the indices after the data has been loaded. These optional methods enable the data loading to the table to be faster. They eliminate the need of consistently updating records in the indices one by one during the data loading.

With the bucket_id partitioned staging table there is no need to update or uphold the bucket_id based indexing and manage it either by the process or by the database engine, as the staging table partitioning key is always updated and managed with the table itself by the database engine.

While investigating and evaluating these different methods (keeping staging table bucket_id index online during data load; dropping bucket_id index prior to data load and recreating bucket_id index after data load; and bucket_id partitioned staging table) against each other, experience and evidence under confidentiality obligations from real life implementations show clear benefits of using the partitioned staging table and AcDWH. If the staging table would not be partitioned, the system would be forced to update the index while inserting the data into the staging table. Given the characteristics of updating the full table index while inserting, the database engine needs to ensure the index is properly updated during the transaction in both cases. This will introduce a delay in the insert operation as the database engine needs to update the indexing structure during the transaction. The database engine would need to search for physical index extents with available space within them or add additional physical extents to the end of index and insert the relevant values to the index. After inserting the rows to the staging table, the insert will be committed.

In general, a commit updates a record perpetually in a database. Within database transaction, a commit saves the changed data permanently and ends a transaction. It also allows other users to see the committed changes. A rollback rolls back the changes after updating the data with changes. As commit, a rollback also ends a transaction and other users will not see any of the changes related to the rolled back transaction. [01]

Looking into the differences of a normal staging table and a partitioned staging table, the key characteristic differences are how large or how many physical extents of structures need to be updated during the insert operation. This applies to the table structures and index structures.

In the case of a normal staging table without table partitioning the index is structured as a normal index having one or more physical extents. Then, there are two approaches how to handle the staging area and also loading the data to the DWH structures; i) handling a batch at a time, having no concurrency; or ii) handling multiple batches at a time introducing concurrency.

In the first case of handling a batch at a time and designing the system for no concurrency, the method is straightforward:

- insert the data to the staging table
- transform and load the data from staging table to the DWH structures
- delete or truncate the staging table after a successfully committed transformation and load transaction to the DWH structures.

If concurrency would be needed for this traditional method, the following additional steps need to be added to the process:

- Add a loading batch identifier to the source data (e.g. a column having the bucket_id or load_id identifier)
- Index the staging table based on the previous identifier column
- Manage the DWH structures loading process with deleting the applied rows with the previous identifier

The concurrency-added method will introduce specific problems:

- The need for database engine managing the identifier index, addressing the global staging table (e.g. the table must be able to keep multiple identifiers)
- The need for database engine managing the data allocation in different extents of the staging table.

AcDWH system with partitioned staging table as proposed with the present work omits the following operations from the system in relation to the staging table and to manage multiple batches of source system data:

- index dropping and recreation
- making index offline and rebuilding the index

The presented AcDWH method facilitates for automatic management of the staging table and staging table identifier index, cleansing of the staging table after successful transformation of the data to data warehousing structures and deletion of the transformed data for any bucket_types.

To sum up, the key differences between traditional DWH and the presented AcDWH method are;

- Different types of data are divided into different bucket_types within the system
- The data for a specific bucket_type is stored in a bucket specific staging table and tablespace
- An AcDWH specific partitioned staging table is utilized for parallel loading and concurrency

The parallelism and concurrency between different bucket_types create more distributed workloads. This is due to data residing in different physical areas and data loading processes accessing separate staging tables concurrently.

5.5 Staging table partitioning in AcDWH

In AcDWH, partitioning of the staging table eliminates the need to index the bucket_id and all I/O operations related to searching the bucket_id index are eliminated. This gives an efficiency boost to the operations loading into or from the staging table. In addition, this enhances the housekeeping process, as purging the loaded data from staging table is done through dropping the partition. The drop partition command does not require any search of indices or updating/recreating indices. The command is DDL and simply removes the partition from the staging table.

For the staging table partitioning, the following basic elements need to be specified:

- What is the maximum value the partition can contain (based on the partitioning key)
- What is the initial physical extent size of the table partition

- What is the additional physical extent size of any additionally needed extent. Additional specifications may include details like what is the tablespace the partition is defined to reside on

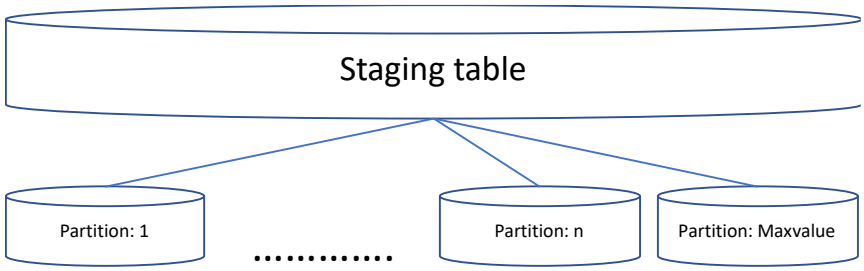

Figure 12. General setup of the partitioned staging table.

Figure 12 shows a general setup of the partitioned staging table structure where there is an arbitrary number of partitions (buckets), illustrated as partitions 1 to n. Additionally the table has a MAXVALUE partition that contains any overflow data, that is not specified to reside in any of the partitions 1 to n. E.g. partition 1 will hold data with bucket_id=1, partition 2 will hold data with bucket_id=2, and so forth. Overflow data would be any data that would get into the staging table due to an error in the AcDWH loading process. The MAXVALUE partition can be continuously monitored for any rows to catch such error situations.

As an example, assuming n=100 a row having bucket_id=101 would be residing in the MAXVALUE partition by definition. The MAXVALUE partition is created to store any inserts with illegal bucket_id and thus avoid catching an error or exception from the database engine within AcDWH system. Any rows with bucket_id column value is greater than n (the highest value defined) are stored in partition MAXVALUE.

Each partition is created with a specification of the maximum value it can hold, e.g. the bucket_id. The partition will hold any rows fulfilling that specification, examples being rows with bucket_id=3 would reside in partition 3, rows for which bucket_id=8 reside in partition 8 and rows for which bucket_id=n reside in partition n.

Each staging table in the setup is initially created with one partition and partitioning key equal to MAXVALUE, see Figure 13.

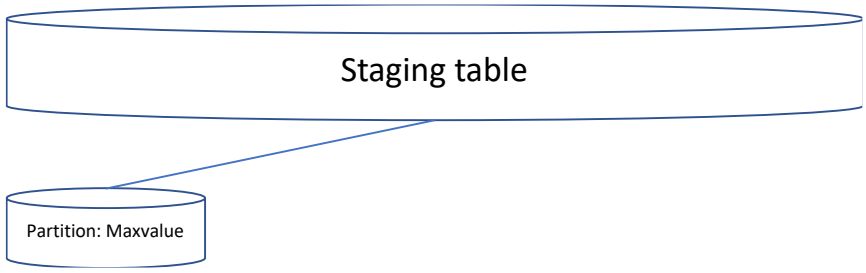

Figure 13. Initial construct of a partitioned staging table in AcDWH.

MAXVALUE represents a value that is always greater than the largest possible value existing in the partitioning key, e.g. an upper bound. MAXVALUE partition acts as the overflow partition catching any non-defined values.

Now, initiating the load from a source system to the staging table, the system will collect the relevant records to the first bucket, bucket_id=1, and generate the partition 1 in the staging table. This is achieved by issuing an alter table SQL language command:

```
ALTER TABLE STAGING_TABLE
SPLIT PARTITION MAXVALUE
INTO (PARTITION PARTITION_0001 VALUES (1),
    PARTITION MAXVALUE VALUES LESS THAN MAXVALUE);
```

The ALTER TABLE SPLIT PARTITION command adds a partition to an partitioned table by splitting an existing partition. There is practically no limit to the number of table partitions. By executing an ALTER TABLE SPLIT PARTITION command, the database engine creates two new partitions (0001 and MAXVALUE) and splits the contents (if any) of the old MAXVALUE partition between the new partitions per the constraints laid out in the partitioning definition; rows with bucket_id=1 to partition 1 and any rows with a larger value to partition MAXVALUE. This is illustrated in Figure 14.

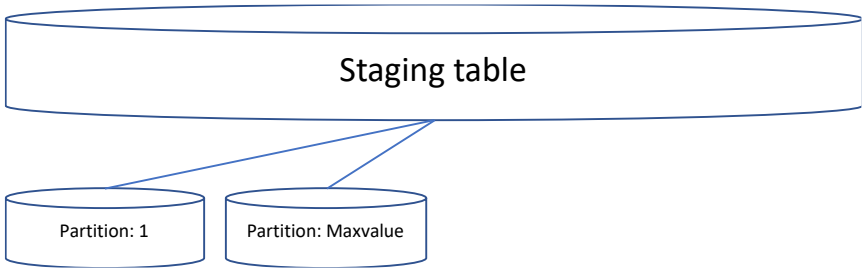

Figure 14. Splitting maxvalue partition into partitions 1 and maxvalue.

The ALTER TABLE SPLIT PARTITION command can include also the tablespace definition to specify in which tablespace the split partition will reside. This will for example give the possibility for a database administrator to spread the staging table physical extents across different storage areas of the DWH system for more I/O throughput. If the tablespace is not defined, the partition will be created in the tablespace defined for the table.

For the use case of this data management system and the initial example above, no indices will be generated for the partitioned staging table. Any INSERT INTO or UPDATE command having WHERE BUCKET_ID=1 will utilize partition pruning. Pruning happens when an SQL operation on a partitioned object is executed. The database engine will recognize the criteria and will address only specific partition(s). This behavior enables the database engine to access only the partitions with relevant data and ignore rest of the partitions. Any database operation addressing a specific bucket_id will be directed only to the partition containing that specific bucket_id. This will isolate the operation from accessing any other partition, limiting the amount of I/O operations and the needed physical object accesses to the specific partition only.

Assuming the system has been running for a while, and inserted *n* new bucket_ids to the partitioned table, the resulting structure is illustrated in Figure 15.

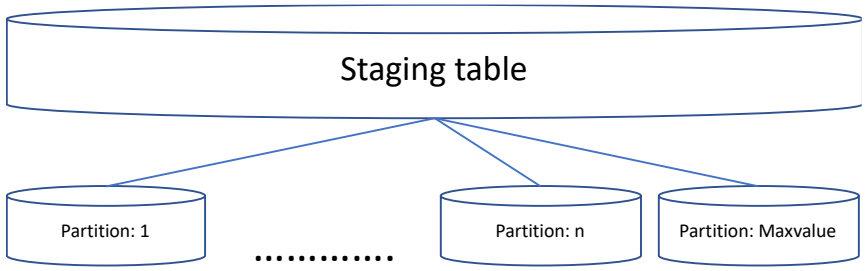

Figure 15. Structure of AcDWH staging table after inserting n buckets.

5.6 Forecasting space requirements, row amounts and generating statistics for the business in AcDWH

For the sake of generating added value from the AcDWH and data gathered during the loading process there are clear areas where AcDWH can be further developed.

The data gathered from the loading process (rows per extent, extent sizes) can be used to generate an automated or a semi-automated system to forecast space requirements and consumption on a specific source area (e.g. a specific staging table). This way the database administrator can forecast and manage the needed storage space for tablespace files for the specific staging tables. This method will remove the potential drawbacks of the halted loading process in case of filled tablespace files.

While AcDWH will map and log the extent sizes of the staging table partitions and their row amounts, the system can be used to report the trend and the prognosis of the needed space for the staging tables.

5.7 Populating the DWH structures

Populating the DWH structures from the staging table is isolated from the previous part of the process in AcDWH, which loads data from a source system and inserts it into the AcDWH staging table. This populating process is the second isolated process area in AcDWH. The structure and construct of using a partitioned staging table is the foundation for the asynchronous

processing of data to staging table simultaneously with processing the data from the AcDWH staging table to DWH structures.

The processing of data from the staging table to AcDWH structures is from process perspective as elegantly simple as loading the staging table. The process will check from the data management system meta data tables, what is the first bucket_id to be handled and the process will then move the rows with found bucket_id from staging table to the AcDWH structures.

The actual processing of data to the AcDWH structures is more complex than the previous part of the input process. This is due to the nature of the complexity of the processing; the processing typically aggregates the data to a specific level, processing will also generate the dimension table details as and if new dimension details are found. Depending on the construct how this part of process is implemented in a particular solution, there are potentially additional elements to be managed. A specific construct is discussed to more detail in chapter 6.1 giving insights into the index management requirements on the referenced technical subject area DWH.

The AcDWH population process has the following process steps:

1. Identify the first bucket_id to be loaded from the staging table
2. Mark the status of bucket_id to “processing” in the bucket metadata table
3. Initialize and launch the AcDWH loading process(es)
4. Update the bucket_id status to “processed” after the AcDWH loading processes for the bucket_id have ended
5. Repeat from step 1, if no new bucket_ids are to be processed, sleep for predefined amount of time (such as one minute) and try again

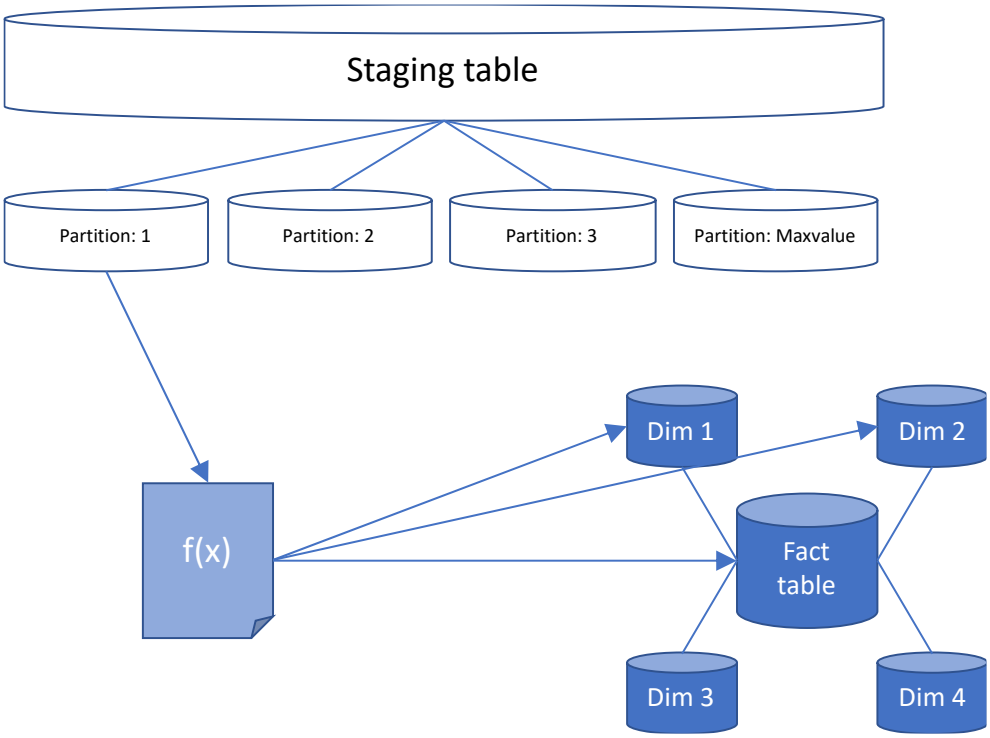

Figure 16. **Processing data from AcDWH staging table partition 1 (bucket_id=1) to AcDWH data warehousing and/or data mart structures. Example shows only data mart example.**

A high level and simplified overview of AcDWH loading process is in Figure 16. The AcDWH loading process is marked as f(x). The process will read the first non-processed bucket_id from the bucket metadata table (for the purposes of this example, bucket_id=1), mark the bucket_id as "processing" to the bucket metadata table, and initialize the DWH load process. When the processing is finished, the process will calculate the time used, how many rows were processed, and how many rows per second were processed and insert the values to the bucket metadata table.

After this, the next scheduled launch of this DWH structure loading process will repeat the process, e.g. read the first non-processed bucket (bucket_id=2) and process as described previously.

5.8 Clearing the AcDWH staging area

The third process part of the AcDWH is independent from the previous process parts (i.e., data loading into staging table and processing data from staging table to DWH structures). The third process part is designed to manage the housekeeping process with the staging table(s). AcDWH manages the staging table's space consumption in an efficient way with a direct access to the specific bucket_id without additional indexing.

5.8.1 Housekeeping process for the staging tables of the AcDWH

The housekeeping process of the AcDWH cleans up the staging table(s). This secures minimal space consumption of the staging table(s) and concurrent processing of the different parts of the AcDWH system. The main intention for the housekeeping process is to eliminate the growing space reservation of the staging table.

The housekeeping process can simply address the purgeable staging table partitions one by one. This is achieved by selecting the lowest BUCKET_ID from the BUCKET_STATUS table where status is 'processed' and altering the staging table by dropping that specific partition.

```
ALTER TABLE STAGING_TABLE DROP PARTITION
PARTITION_||(SELECT MIN(BUCKET_ID) FROM
BUCKET_STATUS WHERE STATUS = 'PROCESSED');

UPDATE TABLE BUCKET_STATUS
SET STATUS = 'DELETED'
WHERE BUCKET_ID=(SELECT MIN(BUCKET_ID) FROM
BUCKET_STATUS WHERE STATUS = 'PROCESSED');
COMMIT;
```

This method issues a data definition language (DDL) command which alters the table structure by dropping the partition and freeing the reserved space for the partition. The system is not able to roll back this operation as it is not a data manipulation language (DML) command.

The speed difference on the DDL and DML operations is phenomenal. While the DML commands generate redo logs and need to be committed, e.g. the rows will actually be deleted, the DDL commands do not generate redo logs and they do not touch the rows of data but rather only execute a command to drop a table partition.

A DDL command executes typically within a few milliseconds and reserves no additional resources from the database when compared to the DML commands where additional resources and space are required.

5.9 The parallelism and concurrency of AcDWH

The AcDWH staging table is constructed by partitioning the staging table to partitions. The cleaning of processed data does not generate extra overhead, nor affect the efficiency of the DWH. Even loading intervals of the data from source systems to the staging table remain unchanged. Similarly, the loading of the data rows from staging table to the AcDWH structures is not affected by the cleaning process. The cleaning process does not generate any overhead for loading processes or cause congestion for the same resources within the database.

Each of the staging table partitions is a separate logical and physical object. Any database operation targeted towards a partition is directly referencing the specific data blocks belonging to that partition. The operation does not access any data blocks belonging to other partitions.

Let us assume staging table would have partitions with bucket_id 1,2 and 3. The setup is illustrated in Figure 17 below.

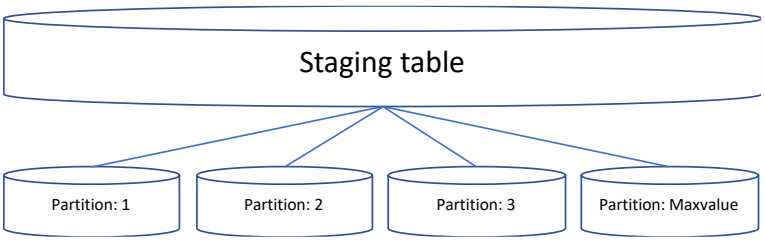

Figure 17. AcDWH staging table having data with bucket_id 1,2 and 3.

Let us assume the following statuses for buckets:

- Bucket_id = 1, bucket_status = ‘PROCESSED’ (data loaded to DWH)
- Bucket_id = 2, bucket_status = ‘PROCESSING’ (data loading to DWH)

- Bucket_id = 3, bucket_status = ‘LOADING’ (data loading to staging table)

Assume that the processing of all three parts of the AcDWH system will be performed in parallel. These processes run independent of each other, concurrent, and the processes are coordinated by the bucket_status metadata table.

The system includes three processes:

1) Staging table loading process – delivers data from source systems to the partitioned staging table
2) DWH loading process – delivers data rows from staging table partition to data warehousing structures
3) Cleansing process – removes staging table rows one bucket at a time

The described behavior isolates the operations from each other, in respect of data access and race for same resources. All partitions reside in physical extents separated from each other.

Process 1) addresses partition 3 with data rows having bucket_id = 3. The process inserts rows from source system(s) to the specific partition 3.

Process 2) addresses staging table partition 2 with data rows having bucket_id = 2. The process selects data rows from the specific staging table partition 2.

Process 3) addresses staging table partition 1 with data rows having bucket_id = 1. The process alters the staging table by dropping the partition. The alter table drop partition command is executed in milliseconds and it does not render rest of the table unaccessible or unusable while the alter table command is executing.

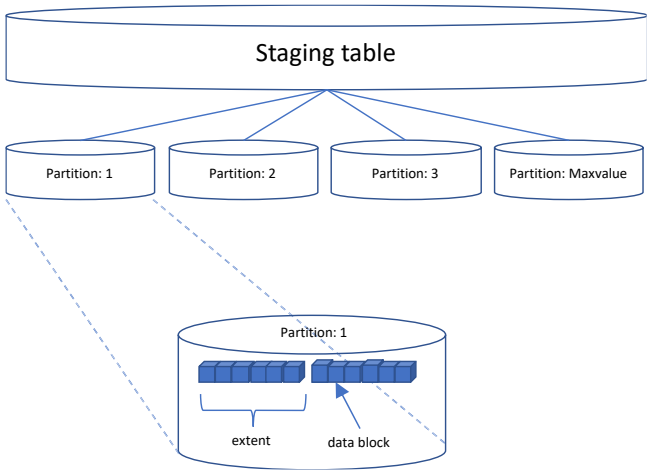

Figure 18. Physical structure of table partition in AcDWH (extents and data blocks).

Considering each partition resides on its own physical extent(s) consisting of data blocks as illustrated in Figure 18, the described structure directs the database engine to access separate areas of database with each system process (process 1-3). These processes can run concurrently without access or congestion on the same data blocks of the data base. This simple method enables the isolation and concurrency of all the process parts. It also provides for asynchronous and independent execution of different process parts using BUCKET_STATUS metadata table.

The independence between the overall process parts is the key element of the method used in AcDWH. A separated physical structure allows the system to remove the staging table partitions independent from other parts. The removal of the partitions by process 3) does not require long execution or wait time or accessing same physical objects with processes 1) and 2). The process 3) is a simple cleaning module, removing processed partitions from the staging table.

If we look onto the disks under the data files, there are multitude of options how the system can be configured for parallelism. The datafiles can be distributed throughout the physical disks. This can be manually achieved by the system administrators, or through the disk system itself. The disks can be configured on the disk system using any level of RAID configuration [32]. Using a RAID configuration, the underlying disk systems can be configured to provide additional parallelism compared to the database level operations.

RAID0 provides striping using the disks defined under that RAID-array, e.g. slicing the physical data across the RAID array disks. RAID0 is however not fault tolerant. RAID1 provides mirroring, any given disk is mirrored with a similar disk having exact copy of the same data. This causes no overhead when writing to the RAID1 disk setup as the same data is written to both disks at the same time. This in change brings a reading performance boost as data can be read from both drives at the same time. There are also advanced RAID configurations (e.g. RAID5 or RAID6/7) that provide fault tolerance with smaller tradeoff in redundant disk space usage. AcDWH has been constructed on systems utilizing different RAID levels ranging from RAID0 to RAID5.

5.10 Logging throughput in AcDWH to analyze operation and process efficiency

The AcDWH includes a method to log the process throughput and process details for further analysis. One use scenario for the process data is to fine tune the size of a data bucket to achieve maximal throughput in the environment in which system is running. While the system is running in production, the BUCKET_STATUS metadata table holds process details such as bucket size and rows per second, which indicates the speed by which the staging table loading process or data warehousing structure loading processes are executing.

This metadata can be used against source system statistical data to secure optimal processing. The system can be modified easily by adjusting the bucket size higher or lower while recording the throughput of the process. The bucket size can be modified to achieve a throughput higher than what the source systems are generating data per each day. This way the system can analyze its operation against the incoming data and any buffering requirements. A buffering requirement would be for example that the data management system can handle 50% more incoming data per day than the source systems generate. By having a buffer in the processing speed the system can keep up with any backlog operations sometimes needed due to connectivity or any other issues getting the data from source systems.

If this reference data is not available, the system cannot adjust its operation and in the worst case the throughput of the system is not keeping up with the

pace of the source system(s) which generate and offload data. This would cause a queuing effect on the data warehousing system, where more and more incoming data is queued in the incoming interface of the system over time due to the system not being capable of processing the incoming data in a required pace.

5.11 Adjusting AcDWH bucket size to enhance throughput

The data bucket size is kept in the system meta data table. The loading processes can be adjusted by changing the data bucket size in the metadata table, each loading process fetches the data bucket size from the system metadata table when they execute.

When the process throughput is analyzed and potentially fine tuning is required as described in previous chapter, the data bucket size can be altered simply through changing this parameter in the system metadata table. The loading processes will take the new data bucket size into use the next time processes execute. After adjusting the bucket size the staging table default extent size might be too small and each partition might consume two extents going forward.

If the space consumption of the staging table starts to be too high, the database administrator can adjust the default extent size to eliminate unnecessary space consumption. The system can be built in a way that it will self-adjust the data bucket size until a maximum throughput is achieved, also automatic tuning of the extent size can be implemented easily.

5.12 Repeatability in AcDWH

One of the key design elements of the AcDWH is repeatability. Repeatability means the ability to process any loading batches again in case of corrupted data on erroneous loading logic. This would mean removing the invalid data from the DWH and loading the batch of data again through AcDWH. The data are organized as a set of buckets which will are managed by the system. As previously described, each bucket of data consists of an

arbitrary amount of records. Due to the nature of this construct, AcDWH system can be enhanced easily to handle repeatability.

In traditional DWH system repeatability is managed by processing a day's material again, if the data has been invalidated by the erroneous DWH transformation logic or invalid source system data. In case of erroneous logic, the transformation logic needs to be corrected, invalid data removed from the DWH and the correct data reloaded. For example, loading one day's data from a source system to a DWH again would mean the following:

- Removing or updating any atomic and aggregate information which is infected by the invalid source system data
- Loading the data identified as invalid or infected by erroneous logic again from source system to staging table(s)
- Transforming the previously infected data again from staging table(s) to DWH structures

Considering the nature of the physical construct of the AcDWH system, securing the repeatability of the transformation and loading processes in case of invalid transformation logic is extremely simple and straight forward.

The AcDWH can be enhanced for example with two simple alternative methods where the incoming source system data will be exported either within the staging table loading process or the house keeping process.

In the first alternative altering the staging table loading process is simple. The source system data can be simultaneously loaded into the staging table and concurrently to a fixed width or delimited text file as a secondary target. This would be an easy method to secure backups of the source system data that can be reloaded into the DWH. This is due to each of the staging table records has the bucket_id identifier as one attribute to of the record. Any of these files including the data for the bucket can be handled through a deviation reload process, where these data files can be reloaded into the staging table. Their bucket_id is then changed to status=loaded and the system will pick the bucket up for DWH processing from the staging table during the next runs of the DWH loading process.

In the second alternative, the housekeeping process can be changed in a simple way: prior to removing the staging table partition with an ALTER

TABLE DROP PARTITION command, the bucket can be exported by the house keeping process either by writing the records to a fixed width or delimited text file as in staging table load process. Or as an alternative by exporting the partition data by using the database engine's data export utility. These table partition exports can be easily imported into the staging table by using the database engine utilities.

Both of these options are similarly easy to implement and provide easy reload capabilities for the AcDWH.

6 Applications of the AcDWH framework

In the present section the concepts of the proposed AcDWH framework are demonstrated by the means of two real-life applications for data warehousing (DWH). Both of these have been architected to use the AcDWH method for active DWH.

Due to the nature of the designed systems only a high-level description of both systems is disclosed.[2] A specific line of business or the real business use scenarios are not disclosed but rather the architecture, business needs and implementation overview are discussed.

6.1 A technical subject area DWH for a specific company *A*

The AcDWH method is in use within company *A*. AcDWH is a platform and data management software used to construct the DWH. The DWH solution uses AcDWH for active DWH to enable continuous loading of the data from source systems to construct a specific technical subject area DWH. The solution is built to analyze technical behavior and error situations in the company *A* technical systems.

A technical element record consists of predefined attributes of a technical events. It typically includes hundreds of fields on the attributes. From a single event at least two records are created, originating record from the outgoing event and the destinating record from the incoming event. In

[2] The examples are real-life systems, that are architected or co-architected by the author. Due to confidentiality obligations, the subject area specific details are not disclosed in this thesis. Examples cover Company *A* and company *B*, which represent different lines of business and different business use scenarios.

addition to the originating and destinating records, business *A* has configured their systems to provide intermittent records from the events, providing additional details when for example either the originating or terminating event transitions to another system segment. In addition to normal event records, the system provides technical records from system components. These records provide technical and statistical details from each system component.

The company *A* gathers all these different records from the system components as flat text files, using fixed width format for the data fields. The record data files are gathered in 10 minutes intervals, and the files are delivered to the company DWH platform. There are separate definitions and system component files for different record data types, examples being records for four different services.

The AcDWH groups the record data files by the record data types into buckets of approximately 50000 records and loads the data into the AcDWH staging tables. There is a similar staging table for each of the record data types and the loading of data is done in parallel for the different record data types.

The records are loaded to the AcDWH structures from the AcDWH staging tables. All these loading processes are run in parallel and AcDWH keeps track of the loaded buckets. The fact tables in the AcDWH structures are partitioned by each hour, e.g. each day has 24 partitions. The records reside in the fact table partitions based on their record timestamp. It has been recorded that in the company the past volumes have been on the range of 100 million records loaded per day. This amounts four million records per hour in an average. An event record has over 500 fields of data, and the average record size is 800 bytes.

Given the details above, the daily source system data volume for the event DWH is of the size 74,5 gigabytes. This amount of data is required to be loaded daily from source systems, transformed to right format and relational model within the event DWH, while, in addition, the technical subject area DWH is simultaneously queried.

The reason for partitioning the fact tables per hour is due to managing the indices. This specific line of business requires to have the fact table refreshed with maximum ten minutes intervals. Any queries from the fact table require indices being in place, otherwise any given query would issue

a full table scan of the fact table. By partitioning the fact tables per hour, the system can isolate the loading and index updating in most cases into a single fact table partition and in the worst case the operations will modify two fact table partitions. The same applies for the index partitions. By partitioning the fact table the system can manage indices being updated while the data is loaded. If the loading window would for some reason extend too long, there is an additional option to alter the partitions of indices offline while inserting the data into the fact table. After the load the index partitions can be rebuilt and made online again.

The technical subject area DWH creates technical summaries of the source system data for the business use. The usages cover for example system capacity planning, peak usage analysis during events and prognosis of system inefficiencies. These analyses are delivered into planning systems to be used as planning data for system development and capacity planning.

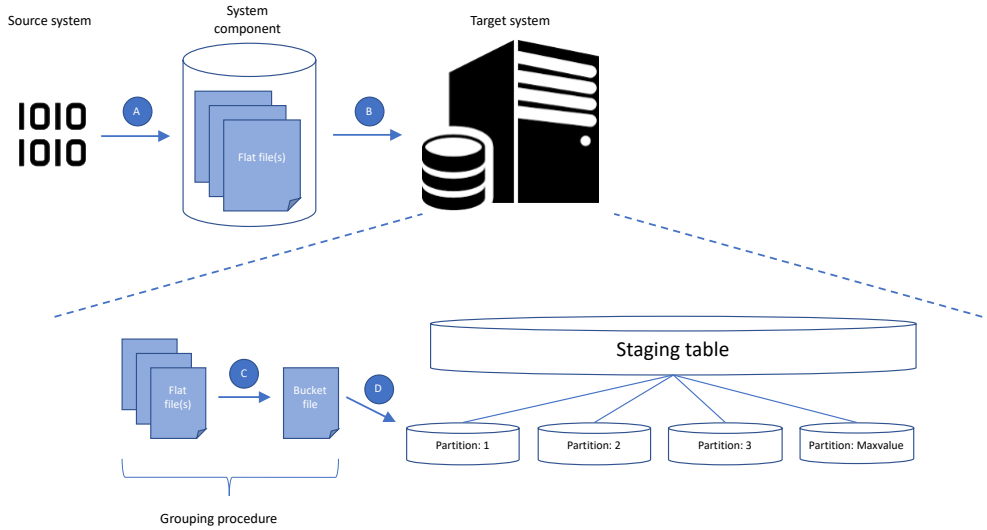

Figure 19. Architecture of the company *A* technical subject area DWH up until to staging area in AcDWH.

Figure 19 describes the architecture of the company *A* technical subject area DWH up until staging table in DWH. The figure is simplified, the system consists of multiple components and devices.

The devices act as the central point in the system setup. All components and devices belong to the same system, and the devices pull the records from the

components periodically; marked as process (A) in the figure. These data files are pushed during the same process by the devices to the technical subject area DWH, marked with (B). The DWH resides in a system segment securely accessible by the devices.

The DWH system assigns the acquired flat files to buckets having at minimum 50000 records in each; marked with (C). This is done by catenating the specific files into one file. From this point onwards the data loading, delivery and cleaning processes are handled by the AcDWH.

Compared to traditional design principles of such DWH, AcDWH removes obstacles in near real-time loading approach in combination with the 24/7 availability for queries and reporting.

6.2 Company *B* data analysis platform

The AcDWH data management system is also in use within company *B* with a slightly modified configuration. Like in description for company *A* technical subject area DWH in chapter 6.1, the solution uses AcDWH to deliver near-real time loading of the data from source systems to the company *B* Transaction DWH (TDWH).

A source system is typically a terminal at company site. The data consists of predefined attributes of a transaction at the site and it includes all the details of the transaction. These include for example the transaction date and time, products and their details as well as the summary of the transaction, potential additional identifiers and the transaction id.

The AcDWH gathers all the transaction details from the sources using a messaging gateway software, such as IBM MQ[3]. The data are pulled by an ETL tool from IBM MQ which gathers the incoming data from sources in 10 minutes intervals.

The AcDWH data management system used by the TDWH groups the data acquired from the MQ system to data buckets of predefined sizes (initially 20000 records). The AcDWH then loads the data into the partitioned staging table and the data are loaded to the TDWH structures from the staging table.

[3] https://www.ibm.com/products/mq

All these loading processes are run in parallel and AcDWH keeps track of the loaded buckets. The fact table partitions in the TDWH structures include data for each day, e.g. each year has 365 or 366 partitions. The data reside in the fact table partitions based on their record timestamps.

The reason for partitioning the fact tables per day is due to managing the indices. The company *B* business requirements are not as time critical as in the case of company *A* business described in chapter 6.1. Queries from the fact table greatly benefit from partitioning of the fact tables because its use speeds up the queries essentially. By partitioning the fact tables per day, the system can isolate the loading and index updating in most cases into a single fact table partition. The same applies for index partitions. By partitioning the fact table to one day partitions, the system can manage the index partitions being altered offline while inserting the data into the fact table. After data load the index partitions can be rebuilt and made online again.

The TDWH of company *B* creates also technical summaries of the source system data for the business use. These include for example terminal usage and transaction volume, which can help company *B* to identify rush hours at the site and plan resources accordingly. These analyses are delivered into resource planning systems to be used as planning data for inventory refresh and employee capacity planning.

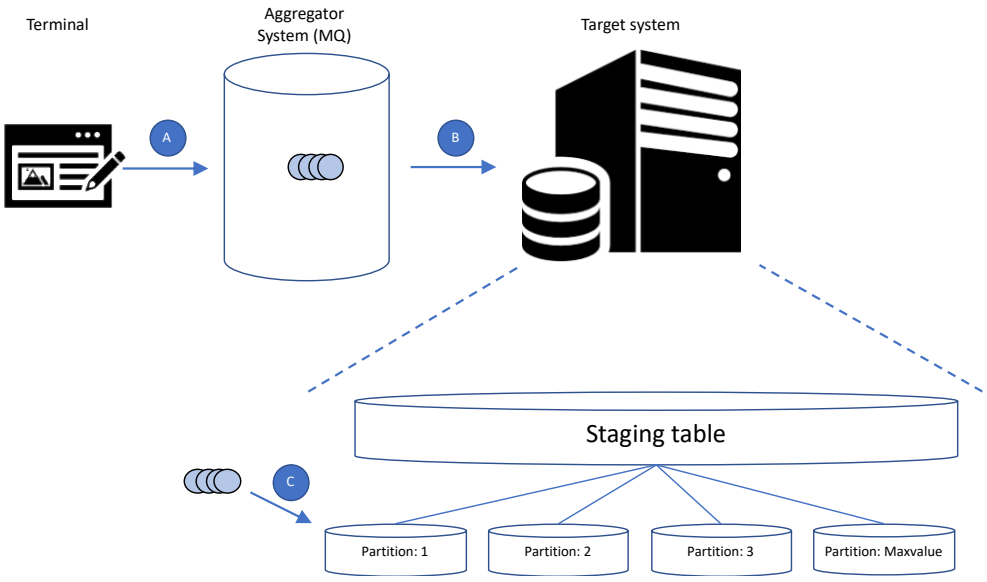

Figure 20. Architecture of the company *B* TDWH up until staging table in AcDWH.

Like in the technical subject area DWH of company *A*, the same basic methodology applies for company *B* TDWH. The data is gathered from the devices (terminals at the company site) by the aggregator system but now via the message queue server; marked as (A) in Figure 20. The aggregator system then offers the records through its message queue interface to the data analysis platform (B). Different from the company *A* case, the data analysis platform itself consumes and requests data from the aggregator system having a message queue server offering the data. The buckets are now formed within the data analysis platform and written onto the AcDWH staging table in 20000 record buckets. When the records are consumed from the message queue and confirmed written to the staging table partition bucket by AcDWH, the message queue client marks the records in the queue as read. The records will eventually get purged from the message queue server itself after new records have been inserted onto the queue.

7 Extensions to the patented AcDWH framework

Two examples of extensions to the AcDWH are presented in this chapter. The first extension deals with the automated data distribution functionality. This extension can enhance the throughput of data loading into partitioned tables. The second extension enhances the AcDWH with an online backup and restore functionality supporting the near real-time operations. The two extensions are examples how the AcDWH can be, due to its constructs, easily extended for additional features and functionalities further enhancing the system.

7.1 Data distribution

This method extends the functionality of the AcDWH system by leveraging the meta data stored within the process and using it to enhance the parallelism & concurrency of a partitioned DWH table to which data are loaded. This method requires the DWH table to be partitioned with hash partitioning.

Hash partitioning is a table partitioning technique where a hash key is used to distribute rows to the table partitions [3]. Hash partitioning can be used in settings where range partitioning isn't naturally usable or appropriate. By using hash partitioning, a row is placed into a partition based on the result of a hash algorithm against the partitioning key. Using the hash partitioning approach, data can be automatically distributed across the table partitions by the database engine. Hash partitioned tables also support partition-wise joins, parallel index access, and the use of parallel data manipulation language (DML).

When using the hash partitioning, the database engine calculates a hash value for the partitioning key column and distributes the records randomly

across the partitions based on the hash value of the partitioning key. This method can be used to enhance throughput by distributing data to more partitions automatically by the database engine. The data redistribution happens automatically by the database engine when a partition is added to a hash partitioned table. When queries address the partitioning key column, the database engine addresses the right partition by the hash value.

For illustration and example;

- a data warehouse (DWH) table is hash partitioned into five partitions
- there is a pre-defined limitation on table loading time per bucket
- data management system loads data into this hash partitioned table
- data management system records the efficiency of aforementioned data loads into the system meta data tables

Using this scenario, the AcDWH can identify a situation where data loading into a hash partitioned table will consume too much time. Long loading times may result from the large number of rows in a partition which mirrors in more time consuming index updates performed online. In this situation the data management system can automatically issue a rearrangement of table partitions by adding additional partitions to the table. The database engine will automatically redistribute the data between partitions in the hash partitioned table. This happens by applying a new hash algorithm to the rows and the database engine relocates the rows into right partitions by the new hash values. This operation is done online and does not require any of the system elements halting the data loading or queries addressed to the table.

For the efficiency reasons, the threshold value for the maximum data loading time per bucket must be defined reasonably low to accommodate for the extra time consumption of the re-distribution of data and online index re-arrangement.

7.2 Near real-time backup and/or restore schematics

The data in near real-time environments is typically changing repeatedly and all the time. This poses large problems for the backup and restore operations of the said near real-time environments. Either the systems need to be periodically in read-only mode or the storage and disk systems need to be

duplicated or mirrored to enable a snapshot of such systems to be taken for backup purposes. Backing up the snapshots will take long time, thus the only viable option in such near real-time loading system is to detach the mirrored disks and take a backup from them.

Restoring the snapshots from detached mirrored disks requires rebuilding the database control files to bring the restored database back online. Neither of the options described are feasible for backing up and restoring near real-time environments.

A simple extension to the AcDWH will provide a method to establish an automated data backup system for DWH tables partitioned by time. The meta data tables of the AcDWH can be added with information on the loaded data. Bucket meta data can be added with information what is the earliest transaction time stamp loaded into the target DWH table. By using this information, the AcDWH can identify which DWH table partitions have been fully populated.

The system can be augmented with a process which will identify these DWH table partitions and execute data offloading into either a fixed width or delimited flat file. Alternatively, the data may be offloaded through a database export utility targeting export of single partition at a time.

In case of corrupted data in the database, any of the exported partitions can be dropped and recreated along with their local indices, and the partition data is easy to load from the partition export. This way the partition indices can be kept online during the loading.

This extension to the data management system would be particularly effective in the company *A* technical subject area DWH case study described in chapter 6.1.

8 Results & Discussion

The proposed AcDWH framework with partitioned staging tables and metadata steered loading system is enhancement over the prior studies in the field for data warehousing (DWH). Prior studies and literature review show that the focus on previous work has been mostly on loading process, join processing and staging area handling. On the other hand, the partitioning of tables has devoted less interest. The prior studies and literature have focused on an industry standard way of handling the staging area with flat files, temporary or in-memory tables, whereas the present study and the proposed AcDWH system use standard database table partitioning functionality in combination with metadata driven loading system.

AcDWH presented in chapter 5 uses a pipelined approach that relies on table partitioning on three main processes: staging table load, DWH load and staging table cleansing.

The AcDWH specific components can be implemented as database procedures and/or external scripts or coding that are executed as pre-source, mid-process or post-target actions within standard ETL tools.

The proposed AcDWH system has the following abilities:

i. it efficiently handles loading parallelism even with standard ETL tools avoiding excessive hand-coded solutions,

ii. eliminates the problems of ever-growing high watermark problem in industry standard non-partitioned staging tables while it uses parallel bucket loading and cleansing processes into the staging table,

iii. enhances the system's throughput while handling parallel insertions, queries and removals of buckets on the staging table,

iv. avoids large extent self-coded systems by utilizing standard functionality in an innovative way, and

v. leaves door open for additional enhancement opportunities, such as data distribution extension and near real-time backup and restore setup of the source data described in chapter 7.

AcDWH with partitioned staging table(s) and metadata steered loading system indicates enhancement over the prior studies and solutions in the field.

The purpose of this study was to identify if a partitioned staging table along with parallelized loading processes would enhance a DWH system's throughput and manageability. This chapter includes discussion of findings related to the previous studies and literature on data loading, staging area handling, ETL processes and table partitioning in a DWH. The chapter includes also discussion of the limitations of the study, areas for future research, and a brief summary.

The key contribution on this study is the enhancement on a DWH system's throughput of data processing pipelines by using the patented AcDWH method two-fold approach: first using a partitioned staging table and secondly parallelizing the DWH loading processes using the staging table as a target or source. This study implicates that utilizing a partitioned staging table enhances DWH loading processes on multiple areas.

The AcDWH method is suitable for the near real-time DWH systems, where the latency of loading data into DWH is of most importance to enable the system to provide as fresh data as possible for the business users. The method proposed in this study enhances the staging area handling considerably with near real-time DWH systems.

The previous studies as outlined in the chapter 4 have discussed areas close to the research question focus area, but none of the previous studies have explicitly studied the usage of partitioned staging table in DWH systems. While Vassiliadis and Simitsis [12] proposed a logical level approach similar to the AcDWH presented in this study, their implementation and study is concentrating only on the logical aspects of the solution. Also, their study does not show how the parallelization of different processes can be enhanced together with the system's throughput if a specifically crafted staging table along with the metadata steered loading system would be in place. In addition to the previous, the proposed AcDWH data management

system indicates advantages on the housekeeping routines of the staging area that further enhance the capabilities of the system.

This study has its limitations since implementation has been tested only using Oracle and IBM database technologies. Other database vendors have not been tested for the implementation and therefore the results on other database vendors are unknown.

9 Conclusions

The key findings of this study are that the use of a traditional staging table is not sufficient for near real-time data warehouse systems or that these systems can be considerably enhanced. Using normal table as a staging table will introduce a) space allocation problems, b) scattered data within the staging table, c) performance problems on staging table handling and d) and process complexity while trying to maintain the performance of the system. The focus of this study was to discuss different staging and data loading process approaches and to also identify if a partitioned staging table along with parallelized loading processes would enhance a DWH system's throughput and manageability. The literature review identified that previous studies had approached the performance and data freshness problems through the process and system setup aspects while none of the previous studies had studied the staging table constructs to a detailed level. Materialized views were discussed in previous studies, along with using different source types such as flat files, external source database tables, message queues and data streams. Previous studies discussed also join processing and data processing.

One of the architectural design aspects behind the data management system is to manage the incoming data flow requirements in conjunction of the cleansing process for efficient staging table handling. In traditional DWH the process manages data loading from source systems throughout the whole process to DWH and reporting structures. The process cannot be optimized and tuned without modifying the processing logic or without adding resources to the system, such as more CPUs, more disk drives on storage system or more network interfaces.

We studied if the partitioned staging table in combination with parallelized loading processes to and from the staging table can help to enhance active DWH systems. This study identified that the process for a near real-time DWH can be materially enhanced and simplified by using partitioned staging table constructs, parallelized loading processes utilizing the partitioned staging table and a metadata driven data loading process, such as the AcDWH. The hypothesis and implementation of the method was tested

through experimental implementation of the two real life systems presented in Chapter 6.
We have tested the AcDWH with two leading technology vendors (Oracle Enterprise Edition and IBM DB2). It should be noted that not all database technology suppliers potentially can provide the specific table partitioning technology that has been used and therefore are not suitable to be used with the AcDWH method.
Future research potential lays with different implementation techniques and options with additional database technology suppliers along with leveraging the same constructs further inside the DWH system to assess if enhancement of data loading and processing can be achieved utilizing same setup. Additionally, the future extensions of data distribution and near real-time backup and restore schematics as described in chapter 7 would potentially be a good focus for future research.

List of References

[1] Building The Data Warehouse. Third edition; WH Inmon; John Wiley & Sons; 2005; http://www.r-5.org/files/books/computers/databases/warehouses/W_H_Inmon-Building_the_Data_Warehouse-EN.pdf

[2] Description of the database normalization basics; Microsoft.com docs; 2020; https://support.microsoft.com/en-us/help/283878/description-of-the-database-normalization-basics

[3] A review on partitioning techniques in database; International Journal of Computer Science and Mobile Computing, Vol. 3, Issue. 5, May 2014, pg.342 – 347; 2014; https://www.researchgate.net/publication/264546464_A_Review_on_Partitioning_Techniques_in_Database

[4] The data warehouse toolkit: The definitive guide to dimensional modeling, third edition; R Kimball & M Ross; John Wiley & Sons, Inc.; 2013; https://books.google.fi/books?hl=en&lr=&id=4rFXzk8wAB8C&oi=fnd&pg=PT18&dq=the+data+warehouse+toolkit

[5] Online table move; J Seifert; US Patent application US2009/0319581A1 - Google patents; 2009; http://patentimages.storage.googleapis.com/pdfs/US20090319581.pdf

[6] On-Demand ELT Architecture for Right-Time BI: Extending the Vision; F Waas, R Wrembel, T Freudenreich, M Thiele, C Koncilia & P Furtado; International Journal of Data Warehousing and Mining; 2013; https://www.igi-global.com/article/content/78285

[7] Data warehousing technologies for large-scale and right-time data; X Liu; PhD Thesis, Aalborg University; 2012; https://orbit.dtu.dk/files/110670162/Data_Warehousing_Technologies.pdf

[8] Near Real-time Data Warehousing Using State-of-the- art ETL Tools; T Jörg & S Dessloch; International Workshop on Business Intelligence for the Real-Time Enterprise; 2009; https://link.springer.com/chapter/10.1007/978-3-642-14559-9_7

[9] Real-time Data Warehouse Loading Methodology; RJ Santos & J Bernardino; IDEAS '08: Proceedings of the 2008 international symposium on Database engineering & applications; 2008; https://dl.acm.org/doi/abs/10.1145/1451940.1451949

[10] Optimizing ETL Processes in Data Warehouses; A Simitsis, P Vassiliadis & T Sellis; 21st International Conference on data engineering; April 2005; https://ieeexplore.ieee.org/abstract/document/1410172

[11] Easy and Effective Parallel Programmable ETL; C Thomsen & TB Pedersen; DOLAP '11: Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP; October 2011; https://doi.org/10.1145/2064676.2064684

[12] Near Real Time ETL; New Trends in Data Warehousing and Data Analysis, pages 1–31; 2009; https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.367.5574&rep=rep1&type=pdf

[13] Near Real-time Data Warehousing with Multi-stage Trickle & Flip; J Zuters; International Conference on Business Informatics Research; 2011; https://www.lu.lv/fileadmin/user_upload/lu_portal/projekti/datorzinatnes_pielietojumi/publikacijas/8_atsk/Zuters_BIR.pdf

[14] An overview of data warehousing and OLAP technology; S Chaudhuri, U Dayal; ACM Sigmod record, 1997; https://dl.acm.org/doi/abs/10.1145/248603.248616

[15] Research problems in data warehousing; J Widom; CIKM '95: Proceedings of the fourth international conference on Information and knowledge management, December 1995; pages 25-30; https://dl.acm.org/doi/pdf/10.1145/221270.221319

[16] Beyond data warehousing: what's next in business intelligence?; M Golfarelli, S Rizzi, I Cella; DOLAP '04: Proceedings of the 7th ACM international workshop on Data warehousing and OLAP; November 2004 Pages 1–6; https://doi.org/10.1145/1031763.1031765 (Cited by 513)

[17] A comparison of data warehousing methodologies; A Sen, A P Sinha; Communications of the ACM; March 2005; https://doi.org/10.1145/1047671.1047673

[18] Data integration in data warehouse; D Calvanese, G De Giacomo, M Lenzerini, D Nardi and R Rosati; International Journal of Cooperative Information Systems, Vol. 10, No. 03; pages 237-271; 2001; https://doi.org/10.1142/S0218843001000345

[19] Real time data warehousing; J J Jonas; US Patent 8,452,787; Google Patents; 2013; https://patents.google.com/patent/US8452787B2/en

[20] ETL queues for active data warehousing; A Karasidikis, P Vassiliadis & E Pitoura; IQIS '05: Proceedings of the 2nd international workshop on Information quality in information systems; June 2005; pages 28–39; https://doi.org/10.1145/1077501.1077509

[21] Method and architecture for automated optimization of ETL throughput in data warehousing applications; S Suresh, J P Gautam, G Pancha, FJ DeRose & M Sankaran; 2001; US Patent 6,208,990; https://patents.google.com/patent/US6208990B1/en

[22] Current practices in data warehousing; R Hackathorn; Bolder Technology, Inc.; November 2002; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.539.8508&rep=rep1&type=pdf

[23] Supporting streaming updates in an active data warehouse; Neoklis Polyzotis ; Spiros Skiadopoulos ; Panos Vassiliadis ; Alkis Simitsis ; Nils-Erik Frantzell; IEEE 23rd International conference on data engineering; 2007; https://ieeexplore.ieee.org/abstract/document/4221696/

[24] Zero-Latency Data Warehousing for Heterogeneous Data Sources and Continuous Data Streams; TM Nguyen & AM Tjoa; iiWAS'2003 - The Fifth International Conference on Information Integrationand Web-based Applications Services; September 2003; https://www.researchgate.net/publication/221237759_Zero-Latency_Data_Warehousing_for_Heterogeneous_Data_Sources_and_Continuous_Data_Streams

[25] Meshing Streaming Updates with Persistent Data in an Active Data Warehouse; N Polyzotis, S Skiadopoulos, P Vassiliadis, A Simitsis & N-E Frantzell; IEEE Transactions on knowledge and data engineering, Volume 20, Issue 7; 2008; https://ieeexplore.ieee.org/abstract/document/4441713/

[26] ETL Evolution for Real-Time Data Warehousing; K Kakish & TA Kraft; 2012 Proceedings of the Conference on Information Systems Applied Research; November 2012; https://www.researchgate.net/publication/280837435_ETL_Evolution_for_Real-Time_Data_Warehousing

[27] Striving towards Near Real-Time Data Integration for Data Warehouses; RM Bruckner, B List & J Schiefer; International Conference on Data Warehousing and Knowledge Discovery 2002, pages 317-326; 2002; https://link.springer.com/chapter/10.1007/3-540-46145-0_31

[28] X_Hybridjoin for near-real-time data warehousing; MA Naeem, G Dobbie & G Webber; British national conference on databases. Advances on databases pages 33-47; 2011; https://link.springer.com/chapter/10.1007/978-3-642-24577-0_5

[29] An Event-Based Near Real-Time Data Integration Architecture; MA Naeem, G Dobbie & G Webber; 12th Enterprise Distributed Object Computing Conference Workshops; 2008; https://ieeexplore.ieee.org/abstract/document/4815048/

[30] Active data warehousing: a new breed of decision support; J Probst; Proceedings. 13th International Workshop on Database and Expert Systems Applications; 2002; https://ieeexplore.ieee.org/abstract/document/1045990/

[31] A Partition-based Approach to Support Streaming Updates over Persistent Data in an Active Data Warehouse; A Chakraborty & A Singh; International symposium on parallel and distributed processing; 2009; https://ieeexplore.ieee.org/abstract/document/5161064/

[32] Introduction to redundant arrays of inexpensive disks (RAID); DA Patterson, P Chen, G Gibson, RH Katz; COMPCON Spring 89; 1989 - computer.org

[33] Fundamentals of Data Warehouses; Jarke, M., Lenzerini, M., Vassiliou, Y.: Panos Vassiliadis; Springer Verlag, 2nd, rev. and extended ed., XIV (2003)

[34] Comparative study of indexing techniques in DBMS; Gupta, M. & Badal, D.; https://www.researchgate.net/publication/333844844_Comparative_study_of_indexing_techniques_in_DBMS

[35] Implementation of database massively parallel processing system to build scalability on process data warehouse; Bani, Fajar Ciputra Daeng, et al.; Procedia Computer Science, 2018, 135:68-79.; https://www.sciencedirect.com/science/article/pii/S1877050918314376

[36] ETL in Near-Real Time Environment: Challenges and Opportunities; Gorhe, Swapnil; no. April, 2020.; https://www.researchgate.net/profile/Swapnil-Gorhe/publication/340938742_ETL_in_Near-real-time_Environment_A_Review_of_Challenges_and_Possible_Solutions/links/5fbe17d892851c933f5812cd/ETL-in-Near-real-time-Environment-A-Review-of-Challenges-and-Possible-Solutions.pdf

[37] Implementation of Data Backup and Synchronization Based on Identity Column Real Time Data Warehouse; Adnyana, I. Gede; Endra Sulastra, I. M. D.; Lontar Komputer: Jurnal Ilmiah Teknologi Informasi, 2020, 11.1: 9.; https://pdfs.semanticscholar.org/d300/befa2333fdfe6de66afb23152827fa17d450.pdf

[38] Efficient incremental loading in ETL processing for real-time data integration; Biswas, Neepa; Sarkar, Anamitra; Mondal, Kartick Chandra; Innovations in Systems and Software Engineering, 2020, 16.1: 53-61.; https://link.springer.com/article/10.1007/s11334-019-00344-4

[39] Timon: A timestamped event database for efficient telemetry data processing and analytics; CAO, Wei, et al.; Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020. p. 739-753.; https://dl.acm.org/doi/abs/10.1145/3318464.3386136

[40] Apache. Hbase. https://hbase.apache.org/, 2008.

[41] Apache. Cassandra. http://cassandra.apache.org/, 2008.

Original Publications

Myllylahti, J, (2017)
European patent specification EP 1 959 359 B1.
European Patent Office Bulletin 2017/47

I

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 959 359 B1

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:
22.11.2017 Bulletin 2017/47

(21) Application number: **07003386.5**

(22) Date of filing: **19.02.2007**

(51) Int Cl.:
G06F 17/30 *(2006.01)*

(54) **A data management system**

Datenverwaltungssystem

Système de gestion de données

(84) Designated Contracting States:
AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IS IT LI LT LU LV MC NL PL PT RO SE SI SK TR

(43) Date of publication of application:
20.08.2008 Bulletin 2008/34

(73) Proprietor: **Tieto Oyj**
00440 Helsinki (FI)

(72) Inventor: **Myllylahti, Jari**
00840 Helsinki (FI)

(74) Representative: **Papula Oy**
P.O. Box 981
00101 Helsinki (FI)

(56) References cited:
EP-A- 1 591 914 **US-A1- 2004 225 664**
US-B1- 7 093 232

- **A.EJAZ ET AL.: "Utilizing Staging Tables in Data Integration to Load Data into Materialized Views" SPRINGER VERLAG, [Online] vol. 3314/2004, 2004, pages 685-691, XP002439036 Berlin, Germany Retrieved from the Internet: URL:http://www.springerlink.com/content/u4a41nnwvttje3hw/fulltext.pdf> [retrieved on 2007-06-25]**
- **MATENA V ET AL: "ENTERPRISE JAVABEANS - DRAFT SPECIFICATION, VERSION 0.9", SUN MICROSYSTEMS SPECIFICATION, XX, XX, no. V0.9, 4 February 1998 (1998-02-04), pages 1-177, XP002424103,**

EP 1 959 359 B1

Printed by Jouve, 75001 PARIS (FR)

Description

FIELD OF THE INVENTION

[0001] The present invention relates to data management systems. Particularly, the invention relates to novel methods, a computer program and a data management system for managing data for example in near real-time environments.

BACKGROUND OF THE INVENTION

[0002] Databases are typically used to store various kinds of data put in a certain format. In order to store data, a solution has to be provided to collect data into the database.

[0003] One a very typical way to collect data into a database system is to use an intermediate data store in the data collection process. Figure 1 discloses one example of a simple data collection and analyzing architecture. A data source 100 is a source that generates data about e.g. transactions. An exemplary transaction could be a sales transaction. A sales transaction occurs e.g. when one pays for his/her purchase at a cash desk. The cash register generates a transaction that includes e.g. time/date, what was purchased and what was the total amount of the purchases. A data source 100 represents the data source that temporarily stores the transaction data. The data source may take any appropriate form of data storage from which it is possible to load data for further processing. The data source 100 is e.g. a database or a flat file.

[0004] A data management system includes one or more loading processes that are arranged to load data from the data source into a temporary data storage area that is often called as "a staging area" or a "staging table" 102. The stating table 102 may, for example, include data relating to all the transactions of a single day. The data is typically indexed in the staging table 102. The loading process loads, for example, a whole day's transactions or for example 10000 transactions from the data source at a time and stores the transactions in the staging table with unique index in case of smaller set of rows than a whole day. Both methods apply. The next set of transactions will be stored after the previously indexed set of transactions with a new index. As transactions are loaded into the staging table, the size of the staging table increases.

[0005] The data management system further typically comprises one or more delivery processes that read data from the staging table and transfer it to a target system 104. When a delivery process has processed a set of transaction relating to a certain index, the data relating to the index has to be deleted from the staging table. This means also that the indexes have to be updated. The actual memory space which was previously allocated for the deleted transaction data relating to the index will not be freed. Therefore, as new transactions are loaded into the staging table, the size of the table increases all the time although some of the transaction data may already have been deleted by the delivery processes. One way to solve this problem is to use a cleaning operation every now and then. Due to the cleaning process, data loading processes and data delivery processes have to be suspended for the duration of the cleaning process and the index has to be rebuilt again.

[0006] One possible target system 104 in Figure 1 is a data warehouse. Previously, every near real time data warehousing project has often been implemented using uniquely designed processes which contain general data loading components trying to gather all the incoming data at particular point of time. Data warehousing processes has been structured in huge mappings. In come cases, even minor changes in one part of the process could result in large process modifications. Furthermore, the staging tables tend to hold data for the whole loading window (typically a day) and the loading addresses the whole period of data which results in long-running huge processes.

[0007] EP-A-1591914 discloses a method, computer program product and device for importing a plurality of data sets into a system. The method comprises providing data sets to be imported on a data storage medium; reading said data sets from said data storage medium; storing said data sets in a first table in said system, substantially without processing said data sets; reading said stored data sets from said first table, preferably data set by data set; and writing one or more data segments of each data set into sub-tables, each sub-table comprising at least one data field, said data field being preferably associated with predefined categories.

[0008] US-B1-7093232 discloses a component stager which accepts from developers one or more versions of a component of a product, and associates with each version a time that is one of a number of the periodically recurring times (also called "ticks"), e.g. an upcoming Wednesday, if the periodically recurring times happen on a weekly basis, i.e. every Wednesday. Such associations may be used to identify one or more versions that are associated with a specific tick, e.g. to identify the amount of work done between to ticks and/or to find versions that are currently available for release, which may be based on a most recent tick, or a tick associated with a milestone in the past.

[0009] The solution disclosed in US 2004/225664 stages data from various sources and performs an automated series of steps on the data to assimilate it into a master SKU table and to make it useable for and available to other applications, including systems for making the information available to a web site hosting an online store. The system includes the use of an abstraction layer to reconcile item identifiers between and amongst various data sources.

[0010] A. Ejaz et al.: "Utilizing Staging Tables in Data Integration to Load Data into Materialized Views", Springer Verlag, vol. 3314/2004, 2004, pages 685-691,

XP002439046 Berlin, proposes an approach to data integration and migration from a collection of heterogeneous and independent data sources into a data warehouse schema.

[0011] Based on the above there is an obvious need for a data management system that would mitigate and/or alleviate the above drawbacks.

SUMMARY OF THE INVENTION

[0012] According to one aspect of the invention, there is provided a computer-implemented method for processing data in a data managing system comprising at least one staging area into which data can be loaded. The following source data loading process loop is executed at least once for a data type of the source data: loading source data from a data source, wherein the loaded amount of source data constitutes a bucket and wherein the bucket comprises at least one row of data; creating a new partition in a staging area for the bucket, wherein the partition is a physical partition in the staging area; determining, based on an identifier scheme, a unique identifier for the bucket; assigning a partition identifier for the partition, the partition identifier comprising the unique identifier of the bucket; tagging each row of data in the bucket with the unique identifier; and storing the bucket in the partition identified by the partitionidentifier.

[0013] The following delivery process is executed at least once for the data type of the source data stored by the loading process: selecting a bucket stored by the loading process in a partition of a staging area; delivering the selected bucket to a target system; and updating in bucket deliverer's metadata that the delivery process has finished processing the bucket.

[0014] The following cleaning process is executed at predefined time intervals for a bucket type, the bucket type in bucket metadata indicating the data type of data in a bucket: checking, from the bucket metadata of a bucket, whether the bucket has been processed by all delivery processes processing the bucket; dropping the partition containing the bucket, when all delivery processes processing the bucket have processed the bucket; and updating in the bucket metadata that the bucket has been dropped.

[0015] According to another aspect of the invention, there is provided a computer program comprising program code configured to perform the method of the invention when executed in a data processing device.

[0016] According to another aspect of the invention, there is provided a data management system comprising at least one staging area in which source data can be stored. The data management system comprises a loading module for loading data in the data managing system, wherein the loading module is configured to execute the following source data loading process loop at least once for a data type of the source data: load source data from a data source, wherein the loaded amount of source data constitutes a bucket and wherein the bucket comprises at least one row of data; create a new partition in a staging area for the bucket, wherein the partition is a physical partition in the staging area; determine, based on an identifier scheme, a unique identifier for the bucket; assign a partition identifier for the partition, the partition identifier comprising the unique identifier of the bucket; and store the bucket in the partition identified by the partition identifier.

[0017] The data management system also comprises a delivery module for delivering data in the data managing system, wherein the delivery module is configured to execute the following delivery process at least once for the data type of the source data stored by the loading module: select a bucket stored by the loading process in a partition of a staging area; deliver the selected bucket to a target system; and update in bucket deliverer's metadata that the delivery process has finished processing the bucket.

[0018] The data management system also comprises a cleaning module for cleaning data in the data management system, wherein the cleaning module is configured to execute the following cleaning process at predefined time intervals for a bucket type, the bucket type in bucket metadata indicating the data type of data in a bucket: checking, from bucket metadata of a bucket, whether the bucket has been processed by all delivery processes processing the bucket; dropping the partition containing the bucket, when all delivery processes processing the bucket have processed the bucket; and updating in the bucket metadata that the bucket has been dropped.

[0019] In one embodiment, the loading module is further configured to generate bucket metadata for the bucket, the bucket metadata determining properties for the bucket stored in the partition. In one embodiment the bucket metadata comprises at least one of the following: bucket identifier, bucket type, number of records in the bucket, earliest record in the bucket, latest record in the bucket, status of the bucket, bucket start time, bucket end time and loaded rows per second.

[0020] In one embodiment, the loading module is further configured to store a duplicate of the bucket into an additional data store during the loading process, and provide the data with the same unique identifier.

[0021] In one embodiment, the bucket size is fixed. In another embodiment, the bucket size is changed at least once.

[0022] In one embodiment, the delivery module is configured to select the bucket based on at least of the following: a bucket identifier; and a time stamp in a bucket metadata.

[0023] In one embodiment, the delivery module is further configured to update the status information of the bucket in the bucket metadata to 'delivering', when starting processing the bucket with the delivery process.

[0024] In one embodiment, the delivery module is further configured to update bucket deliverer's metadata, when the delivery module starts processing the bucket, the updating comprising: adding a deliverer identifier in

the bucket deliverer's metadata, and adding a deliverer start time in the bucket deliverer's metadata.

[0025] In one embodiment, the delivery module is further configured to update the bucket deliverer's metadata, when the delivery module ends processing the bucket, the updating comprising: adding a deliverer end time in the bucket deliverer's metadata.

[0026] In one embodiment, the delivery module is further configured to check, whether the bucket deliverer's metadata comprises other deliverer identifiers identifying delivery processes that process the same data type, check, whether there exists a deliverer end time for each deliverer identifier, and update the status information in the bucket metadata to 'delivered', when there exists a deliverer end time for each deliverer identifier.

[0027] In one embodiment, the data management system comprises delivery process metadata for each delivery process, the metadata comprising at least one of the following: an identifier of the delivery process, a name of the delivery process, a type of the delivery process, a description for the delivery process, priority of the delivery process, and a target bucket type of the delivery process.

[0028] In one embodiment, a priority is set to at least one delivery process.

[0029] In one embodiment, the amount of simultaneously executed delivery processes is limited.

[0030] In one embodiment, the properties of at least one delivery process are changed.

[0031] In one embodiment, the amount of delivery processes is changed.

[0032] In one embodiment, the cleaning module is further configured to check, whether the bucket deliverer's metadata comprises other deliverer identifiers identifying delivery processes that process the same data type; check, whether there exists a deliverer end time for each deliverer identifier; and update the status information in the bucket metadata to 'delivered', when there exists a deliverer end time for each deliver identifier.

[0033] In the invention, all delivery processes may run in parallel and are isolated from each other. Therefore, the delivery processes can address the same bucket simultaneously, which gives the system ability to be parallelized and scaled up or out when performance boost is needed.

[0034] The advantages of the invention relate to improved source data handling stored in the staging area. For example, the loading process may be executed independently from the delivery process(es). Correspondingly, the delivery process(es) may be executed independently from the loading process as long as there are buckets in the staging table that are not processed yet.

BRIEF DESCRIPTION OF THE DRAWINGS

[0035] The accompanying drawings, which are included to provide a further understanding of the invention and constitute a part of this specification, illustrate embodiments of the invention and together with the description help to explain the principles of the invention. In the drawings:

Figure 1 one example of a simple prior art data collection and analyzing architecture;

Figure 2 discloses a general model of a data management system according to one embodiment of the invention;

Figure 3A explains the operation of a loading process in a data management system according to one embodiment of the invention;

Figure 3B one possible form of bucket metadata according to one embodiment of the invention;

Figure 3C explains the operation of a loading process in a data management system according to another embodiment of the invention;

Figure 4 discloses metadata relating to a delivery process according to one embodiment of the invention;

Figure 5A explains the operation of a delivery process in a data management system according to one embodiment of the invention;

Figure 5B discloses bucket deliverer's metadata according to one embodiment of the invention;

Figure 5C explains the operation of a delivery process in a data management system according to another embodiment of the invention;

Figure 5D explains the operation of a delivery process in the data management system according to another embodiment of the invention;

Figure 6A discloses a partition cleaning process according to one embodiment of the invention;

Figure 6B discloses a partition cleaning process according to another embodiment of the invention;

Figure 7 discloses an example of prioritization of delivery processes according to one embodiment of the invention; and

Figure 8 discloses a data management system according to one embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0036] Reference will now be made in detail to the embodiments of the present invention, examples of which are illustrated in the accompanying drawings.

[0037] Figure 2 represents a general model of a data management system according to one embodiment of the invention. The model includes at least one data source 200 that is e.g. a database, a flat file, an xml file, message queue etc. Data is read from the data source 200 into a staging area 202 (or staging table) in variable row amounts. The amount of rows loaded from the data source at a time may be e.g. fixed on a system level. In another embodiment, the amount of rows may be variable and it can be determined upon starting a loading process 206 e.g. from a parameter or a parameter file. The concept of varying the amount of rows to be loaded from the data source will be discussed in more detail later.

[0038] A term "bucket" or "bucketing" will be used throughout the description to represent the amount of data (e.g. data rows) loaded from the data source 200 at a time.

[0039] When a bucket has been read from the data source 200, the data management system determines a unique identifier for the bucket. In one embodiment, the identifier is unique only among the buckets of the same data type. In another embodiment, the identifier is unique among all buckets of all data types. When the unique identifier has been determined for the bucket, the data management system creates a new partition 210 in the staging table 202 for the bucket. At the same time, the partition may be given an identifier. Let's assume that the unique identifier for the bucket would be *00000001*. In this case, the identifier for the partition could be e.g. *stage cust123trans p00000001*. *Cust123* identifies the customer and *trans* identifier the data type (in this case, transaction). P refers to partition and *00000001* is the bucket identifier. Therefore, in one embodiment, the partition identifier directly identifies also the bucket identifier. If the partition identifier does not directly identify also the bucket identifier, in one embodiment, there is a mapping table that creates a mapping between the partition identifier and the bucket identifier.

[0040] The data stored in the partitions 210 in the staging table 202 is processed by one or more delivery processes 208. Independently of the loading processes 206 that load data to the stating table 200 from the data source(s) 200, the delivery processes 208 process data stored in the partitions 210. The delivery processes deliver data to target system(s) 204, e.g. to a data warehouse population, data mart population or external interfaces or authorities. There may be several delivery processes that are run in parallel manner.

[0041] The loading processes 206 and delivery processes will be discussed in more detail shortly.

[0042] The principal idea in the data management system disclosed in Figure 2 is that data is loaded from the data source(s) 200 into separate partitions 210 in the staging table 202. Due to this fact, there are no data locking problems while the data is processed since each partition is a separate physical structure.

[0043] Furthermore, in yet another embodiment of the invention there are further advantages. The processes are implemented with asynchronous process modules. Therefore, the modules are detached from each other and they communicate only through common administrative data model or data layer. In other words, the loading processes 206 and delivery processes 208 operate independently from each other. Furthermore, all delivery processes may be run in parallel and they may be isolated from each other. Each delivery process can address the same bucket simultaneously which gives the system ability to be parallelized and scaled up or out when performance boost is needed.

[0044] Figure 3A explains the operation of a loading process in the data management system according to one embodiment of the invention. The operation is explained together with Figure 3B which represents one possible alternative for bucket metadata. The bucket metadata 320 stores information about the bucket. The term 'metadata' is used throughout the description to refer to system information that determines general properties for e.g. data buckets and delivery processes.

[0045] When a data source starts to provide data (step 300), a new partition is created for the data (step 302) in a staging table. Before the partition is created for the data, it may be necessary to create a zero partition in the staging table. The zero partition is never touched and therefore there is always one partition (zero partition) in the staging table. In the following a term "bucket" refers to a certain amount of source data that is loaded at a time from the data source. The bucket size may be predetermined (e.g. 50000 rows) . In other words, it may be fixed on system level. It may also be variable called upon stage load process launch via parameter or parameter file or it may be decided on performance metrics from metadata on process launch time. An identifier is determined for the bucket (step 304). In one embodiment, the identifier is unique only among the buckets of the same data type. In another embodiment, the identifier is unique among all buckets of all data types. The data management system comprises a control bucket 310 (in other words, a control table or bucket metadata) which stores various pieces of information about the bucket. When the partition has been created and the identifier determined, the loading process updates the bucket metadata 320 and add a new entry (i.e., a new row) for the bucket. The following pieces of information are added or updated for the bucket in the bucket metadata:

- *BUCKET_ID* (326). E.g. a sequential number starting from 1. Each bucket has a unique identifier (e.g. number). In one embodiment, if the identifiers are sequentially allocated, the identifiers also determine the loading order of buckets into the stating table. In other words, a bucket having a smaller identifier than that of another bucket has been loaded and stored earlier.
- *BUCKET TYPE* (328). BUCKET TYPE defines the type of the bucket, for example, 'transaction', 'customer data' etc.
- *STATUS (created)* (336). Current status of the bucket, for example, 'created', 'loading', 'delivering', 'delivered', 'deleted' etc.

[0046] When the data loading actually starts (step 306), the actual data rows may be tagged with the bucket identifier (e.g. there is added a bucket_id column to the staging table) and the following pieces of information are added or updated for the bucket in the bucket metadata:

- *STATUS (loading)* (336).
- *BUCKET STARTTIME* (338). This information indicates a timestamp of the creation of the bucket.

[0047] When the bucket has been loaded, the following pieces of information are updated for the bucket in the bucket metadata:

- *NUMBER OF ROWS* (330). The number of rows in the bucket.
- *EARLIEST_RECORD* (332). Timestamp of the earliest record/transaction in the bucket.
- *LATEST RECORD* (334). Timestamp of the latest record/transaction in the bucket.
- *STATUS* (loaded) (336). Current status of the bucket.
- *BUCKET ENDTIME* (340). Timestamp after finalizing the bucket (all data has been loaded in the partition).
- *ROWS_PER_SEC* (342). The number of rows loaded in the staging table per second.

[0048] The column 322 refers to the used variables in general and the column 324 indicates a variable type of each variable. Furthermore, although Figure 3A represents that bucket metadata is updated after certain steps, it is also possible to update the bucket metadata in different process stages than disclosed in Figure 3A. Furthermore, Figure 3B discloses only one embodiment of possible bucket metadata, and therefore in other embodiments the bucket metadata may differ from the bucket metadata disclosed in Figure 3B. For example, at least one of ROWS_PER_SEC, BUCKET_STARTTIME and BUCKET_ENTTIME may be missing in other embodiments or there may be some additional metadata entries.

[0049] In one embodiment of the invention, bucketing (that is, the amount of rows) is done initially based on an initial value. Later on based on e.g. the metrics collected from stage loading and data delivery operations may change the initial value. Furthermore, bucket metadata may contain also performance metrics which can be utilized continuously on future bucket sizing functions for best performance (for example, auto throttling, the loading system can adjust itself based on the collected metrics for best possible throughput, central processing unit utilization etc.).

[0050] Summarizing the above, buckets contain 1 to n rows of data (variable bucket size) and each bucket is represented by a partition on the staging table. In one embodiment, if there multiple data types, each different data type uses a dedicated staging table. Partitioning the staging table (e.g. a range or a list) is done e.g. based on the bucket identifier which is unique either inside each data type or the whole system. Furthermore, the bucketing also isolates the reading operations. Partitioning the data by buckets enables the data delivery processes to address only the rows of the distinct bucket without indexing the tables. With this structure the staging tables do not have to have any indexes to give performance on bucket reads. Actually, indexing on staging tables may bring lower performance on populating the staging tables on initial load.

[0051] Figure 3C explains the operation of a loading process in the data management system according to another embodiment of the invention.

[0052] When a data source starts to provide data (step 360), a new partition is created for the data (step 362) in a staging table. Before the partition is created for the data, it may be necessary to create a zero partition in the staging table. The zero partition is never touched and therefore there is always one partition (zero partition) in the staging table. The data is then stored in the created partition. The same loading process repeats when new data is ready to the loaded into the staging table. The loading process again creates a new partition for the new data and stores the data in the new partition (step 364). The partitions are preferably identified e.g. by sequential numbers. The next set of source data is stored in a partition whose identifier is greater than that of the previous partition. Therefore, the order of the partition identifies also the loading order of the source data in the partitions.

[0053] Figure 4 discloses possible metadata 400 relating to a delivery process according to one embodiment of the invention. A delivery process is a process that reads partitions in a staging table and delivers the data (rows) from the partitions to a target system or target systems. The target system may include e.g. one of the following: a data warehouse structure population, a data mart population, external interfaces and authorities.

[0054] The data management system may comprise built-in delivery processes. In another embodiment, it is possible to define new delivery processes as the need arises. The amount of delivery processes is not limited to any certain amount. There can be as many delivery processes as the data management system needs to have. In one embodiment, delivery processes may be prioritized. In other words, for example in one case a data warehouse structure population must be done before any other delivery process is allowed to touch the same bucket. In another embodiment, all the delivery processes stand on the same line and they all can run in parallel.

[0055] In the embodiment of Figure 4, each delivery process has been assigned the following metadata:

- *DELIVERER ID* (402). Unique identifier for the deliverer (i.e. delivery process).
- *DELIVERER NAME* (404). Name of the delivery process, for example, 'STORAGE', 'DATAMART_SALES', 'DATAMART_OUTLET' etc.
- *DELIVERER_TYPE* (406). Type of the delivery process, for example, 'DW', 'Datamart', 'Interface' etc.
- *DELIVERER_DESCRIPTION* (408). Short description of the delivery process, for example, 'DW population', 'Sales datamart population', 'Interface 1 to authorities' etc.
- *PRIORITY* (410). Priority of the delivery process.
- *DELIVERER TARGET* (412). Type of the data that the delivery process delivers, for example, 'customer data'. This variable links to BUCKET_TYPE variable in the bucket metadata.

[0056] Figure 4 discloses only one embodiment of possible deliverer metadata, and therefore in other embodiments the deliverer metadata may differ from the metadata disclosed in Figure 4. For example, in other embodiment some metadata entries may be missing or there may be some additional metadata entries.

[0057] Figure 5A explains the operation of a delivery process in the data management system according to one embodiment of the invention. The operation is explained together with Figure 5B which represents one possible alternative for bucket deliverer's metadata 520. The bucket metadata stores information about the bucket.

[0058] A delivery process selects a bucket from the staging area (step 500). The delivery process may select the bucket by determining the smallest bucket identifier which points to a bucket that has not yet been processed. In another embodiment, the bucket selection may be made based on a timestamp in the metadata relating to the bucket. The timestamp used may e.g. be the one that identifies when the bucket has been created. Yet in another embodiment, the selection may be made based on both the bucket identifier and the timestamp. This may be the case e.g. when the sequential identifier "turns" over, that is, starts again from number one. In this case, the bucket having a bucket identifier "1" is not the earliest one since the bucket numbering has started again from the beginning. Therefore, the delivery process may also use the timestamp value in the bucket metadata in determine which one of the buckets is the earliest one. In this example, the buckets are processed in the same order that they were stored in partitions in the loading process. One way to implement is that a sequential identifier is assigned for each bucket. When comparing the identifier, a smaller identifier refers to an earlier bucket.

[0059] The delivery process opens the partition holding the data of the bucket identifier (step 502) . When the status of a bucket in the bucket metadata is 'loaded', the bucket is waiting for the delivery process to start processing the bucket. 'Loaded' means that the bucket has been successfully loaded into the staging table. When the delivery process starts to process the determined bucket, the status of the bucket is updated as 'delivering' in the bucket metadata (step 504). If the status of the bucket is already 'delivering', it means that that are several different delivery processes that may process the bucket at the same time. Furthermore, if the status of the bucket is already 'delivering', the delivery process leaves the bucket metadata untouched.

[0060] When the delivery process starts to deliver data rows from the staging table, each delivery process processing the bucket creates (step 510) a new row with the following pieces of information in the bucket deliverer's metadata 520:

- *BUCKET_ID* (522). E.g. a sequential number starting from 1. Each bucket has a unique identifier (number). In one embodiment, if the identifiers are sequentially allocated, the identifiers also determine the loading order of buckets into the stating table. In other words, a bucket having a smaller identifier than that of another bucket has been loaded and stored earlier.
- *DELIVERER_ID* (524). Deliverer identifier.
- *DELIVERER_STARTTIME* (526). Timestamp, when the deliver started processing.

[0061] Next, the delivery process starts to deliver data rows from the bucket (partition) (step 506). When the deliverer has delivered all the data (rows) contained in the bucket, the bucket deliverer's metadata is again updated (step 510) and the partition is dropped (step 508):

- *DELIVERER_ENDTIME* (528). Timestamp, when the deliver ended processing.
- *ROWS_PER_SEC* (530). The number of rows delivered per second.

[0062] The column 532 refers to the used variables in general and the column 534 indicates a variable type of each variable. The length disclosed after each variable type gives one possible length and thus also other lengths can be implemented. Furthermore, although Figure 5A represents that the bucket metadata is updated after certain steps, it is also possible to update the metadata in different process stages than disclosed in Figure 5A. Furthermore, Figure 5B discloses only one embodiment of possible bucket deliverer's metadata, and therefore in other embodiments the metadata may differ from the metadata disclosed in Figure 4. For example, in other embodiment some metadata entries may be missing or there may be some additional metadata entries.

[0063] Figure 5A disclosed an embodiment in which a delivery process does not itself update the status information in the bucket metadata as 'delivered'. Figure 5C discloses another embodiment in which the last delivery process processing a bucket (partition) also updates the bucket metadata.

[0064] The processing starts from step 508 of Figure 5A. The delivery process checks from the deliverer metadata (see Figure 4) how many delivery processes are assigned to process this particular bucket type (step 540). Next the delivery process checks from the bucket deliverer's metadata the number of delivery process end times for the bucket type (step 542). If the number of delivery processes assigned to process the bucket type is the same as the number of end times in the bucket deliverer's metadata (step 544), it means that all the delivery processes assigned to process the bucket type have already performed their processing. Therefore, the delivery process updates the status information in the bucket metadata as 'delivered' (step 548). If the number of delivery processes assigned to process the bucket type is not the same as the number of end times in the bucket deliverer's metadata, it means that at least one delivery process has not done its processing on the buck-

et.

[0065] Figure 5D explains the operation of a delivery process in the data management system according to another embodiment of the invention. In this embodiment, the data management system comprises only one delivery process.

[0066] The delivery process selects from a staging table a partition that contains the earliest source data (step 560). The earliest source data is determined e.g. by partition identifiers. In one embodiment, the partition having the smallest partition identifier contains the earliest source data. In other words, the delivery process processes the data in the staging table by the "first in first out" principle. The oldest partition is processed first. When the oldest partition has been determined, source data stored in the partition is delivered to at least one target system (step 562). When all the data contained in the partition has been delivered, the partition is deleted (step 564).

[0067] Figure 6A discloses a partition cleaning process according to one embodiment of the invention. The partition cleaning process disclosed in Figure 6A is not tied to the earlier disclosed loading and delivering process. In other words, the partition cleaning process operates as an independent demon process.

[0068] When the processing starts, the cleaning process checks the status information of a bucket in the bucket metadata (step 600). If the status information of the bucket is 'delivered' (step 602), it means that all delivery processes have performed their processing. Therefore, when the status information of the bucket is 'delivered', the partition comprising the bucket is deleted (step 604). After that the status information for the bucket in the bucket metadata is updated to 'deleted'. The processing advances to step 608 in which the cleaning process starts the same processing for a next bucket. If the status information of the bucket was not 'delivered' (step 602), the processing advances directly to step 608 in which the cleaning process starts the same processing for a next bucket.

[0069] In one embodiment of Figure 6A, the cleaning process in run at predetermined intervals. The interval is determined e.g. by a system parameter. The parameter may e.g. determine that the cleaning process in executed e.g. once in an hour, in two hours, in six hours, in one day or any other appropriate interval.

[0070] Figure 6B discloses a partition cleaning process according to one embodiment of the invention. The partition cleaning process disclosed in Figure 6B is not tied to the earlier disclosed loading and delivering process. In other words, the partition cleaning process operates as an independent demon process.

[0071] When the processing starts, the cleaning process checks the status information of a bucket in the bucket metadata (step 620). If the status information of the bucket is 'delivered' (step 622), it means that all delivery processes have performed their processing. Therefore, when the status information of the bucket is 'delivered', the partition comprising the bucket is deleted (step 624). After that the status information for the bucket in the bucket metadata is updated to 'deleted' (step 626). The processing advances to step 628 in which the cleaning process starts the same processing for a next bucket. If the status information of the bucket was not 'delivered' (step 622), the processing advances directly to step 628 in which the cleaning process starts the same processing for a next bucket.

[0072] When the bucket metadata has been updated in step 626, the cleaning process checks from the deliverer metadata (see Figure 4) how many delivery processes are assigned to process this particular bucket type (step 628). Next the cleaning process checks from the bucket deliverer's metadata the number of delivery process end times for the bucket type (step 630). If the number of delivery processes assigned to process the bucket type is the same as the number of end times in the bucket deliverer's metadata (step 632), it means that all the delivery processes assigned to process the bucket type have already performed their processing. Therefore, the cleaning process updates the status information in the bucket metadata as 'delivered' (step 634). If the number of delivery processes assigned to process the bucket type is not the same as the number of end times in the bucket deliverer's metadata, it means that at least one delivery process has not done its processing on the bucket. If a delivery process has not finished its processing (and the status in the bucket metadata is still 'delivering'), the cleaning process continues to step 628 in which the cleaning process starts the same processing for a next bucket.

[0073] In one embodiment of Figure 6B, the cleaning process in run at predetermined intervals. The interval is determined e.g. by a system parameter. The parameter may e.g. determine that the cleaning process in executed e.g. once in an hour, in two hours, in six hours, in one day or any other appropriate interval.

[0074] By utilizing bucketing via range or list partitioning it is possible to ensure that the performance of the staging tables is the highest possible. There will be no issues of rising high watermarks or staging table rebuilds as the deletion of rows of a bucket addresses only a table partition. An actual bucket deletion (when all delivery processes processing a bucket have finished) happens via table partition dropping, not by deleting rows. The bucket/partition cleanup process will run, in one embodiment, asynchronously with delivery processes and operations of the cleanup process are done based on administrative metadata from process status tables.

[0075] Figure 7 discloses an example of prioritization of delivery processes according to one embodiment of the invention. When a delivery process start processing a bucket (a partition) (step 700), it checks priority information relating to the delivery processes in general (step 702). For example, a system designer may have prioritized some delivery processes before others. A first delivery process, for example, may have a priority 01 in its

priority field (step 704). At the same time, a second delivery process may have a priority 02 in its priority field. This means, for example, that the first delivery process has to be finished before the second delivery process may start processing the same bucket (step 706). The first delivery process may determine, for example, from the bucket deliverer's metadata (see Figure 5B) whether the second delivery process has finished its processing with the bucket. If, in this case, there is an end time updated in the bucket deliverer's metadata for the second delivery process (step 708), the first delivery process is now able to may proceed with its bucket processing (step 710).

[0076] If all the delivery processed have the same priority or there has not been assigned priorities to the delivery processes at all, the delivery process may proceed with its bucket processing (step 710).

[0077] The above prioritization example was described by using only two delivery processes. It is evident to a man skilled in the art that there may be more than two delivery processes which have been prioritized based on some prioritization scheme.

[0078] Figure 8 discloses a data management system 808 according to one embodiment of the invention. The data management system comprises a loading module 800 for loading data in a data managing system. The loading module 800 is configured to execute the following source data loading process loop at least once for a data type of the source data: load source data from a data source, wherein the loaded amount of source data constitutes a bucket; determine a unique identifier for the bucket based on an identifier scheme; create a new partition in a staging area 806 for the bucket; and store the bucket in the partition.

[0079] The data management system further comprises a delivery module 802 for delivering data in a data managing system. The delivery module 802 is configured to execute the following delivery process at least once for a data type of the source data: select a bucket from the staging area 806, wherein the staging area 806 comprises at least one partition, each partition storing a bucket comprising source data loaded from at least one data source; deliver the bucket stored in the partition to a target system; and update bucket delivers metadata.

[0080] The data management system may also comprise a cleaning module 804 for cleaning at least one staging area 806 in the data managing system. The dashed line of the box 804 means that the cleaning module may be an optional feature. Further, the cleaning operations may also be executed manually e.g. by an administrator. If there is a cleaning module in the data management system, the cleaning module 804 is configured to execute the following cleaning process at predefined time intervals for a bucket type: check, whether the status of a bucket is 'delivered' in a bucket metadata; delete the partition, from the staging area 806, containing the bucket; and update the status of the bucket to 'deleted' in the bucket metadata.

[0081] The embodiment described above discloses that the cleaning process is an automatic process. In another embodiment of the invention, the cleaning process may be executed manually. Needed actions are executed e.g. by a database administrator.

[0082] In another embodiment of Figure 8, the loading module may be simpler than disclosed above. When a data source starts to provide data, the loading module 800 is configured to crate a new partition in the staging area 806. Before the partition is created for the data, it may be necessary to create a zero partition in the staging table. The zero partition is never touched and therefore there is always one partition (zero partition) in the staging table. The data is then stored in the created partition. The operation of the loading module repeats when new data is ready to the loaded into the staging area 806. The loading module again creates a new partition for the new data and stores the data in the new partition. The partitions are preferably identified e.g. by sequential numbers. The next set of source data is stored in a partition whose identifier is greater than that of the previous partition. Therefore, the order of the partitions identifies also the loading order of the source data in the partitions.

[0083] When source data is to be delivered, the delivery module 802 is configured to select from the staging area 806 a partition that contains the earliest source data. The earliest source data is determined e.g. by partition identifiers. In one embodiment, the partition having the smallest partition identifier contains the earliest source data. In other words, the delivery process processes the data in the staging table by the "first in first out" principle. The oldest partition is processed first. When the oldest partition has been determined, the delivery module 802 is configured deliver the stored source data to at least one target system. When all the data contained in the partition has been delivered, the delivery module 806 is configured to delete the partition.

[0084] In one embodiment of the invention, it is possible to limit the number of simultaneously performed delivery processes. This can be achieved, for example, with a system metadata that determines how many delivery processes can be executed simultaneously.

[0085] In one embodiment of the invention, simultaneously with populating the bucket partition the data may be loaded into, for example, flat files for archival, backup and possible reload purposes. This way it is possible to hold only the needed amount of buckets in the staging table and fetch the older buckets from a file system if necessary.

[0086] In one embodiment of the invention, the invention enables also the reload of a bad data bucket (bucket containing illegal or corrupted data from a source system or changed processing logic in delivery process). The data can easily be identified by its bucket identifier if bad data is loaded, for example, to data warehouse structures, data marts or external interfaces. Deletion of the data and reloading the bucket will be fast as the data structures in the data warehouse, data marts or external

interfaces will be indexed also based on the bucket identifier. If buckets are kept in the staging table for a day or more, the possible reloading is much easier as the reload data already resides in the database structures (the staging table) and there is no need to fetch it from the backup files.

[0087] In one embodiment of the invention, the invention is able to provide an auto throttle feature. The data management system may monitor what is, for example, the relationship between throughput and bucket size. Based on the monitoring, it is possible to change the bucket size dynamically based on various emphasizes, for example:

- the creation of buckets is as fast as possible
- the creation of buckets is as fast as possible and the delivery processes work as fast as possible
- the delivery processes work as fast as possible.

[0088] In other words, the data management system may optimize the bucket size based on different emphasizes.

[0089] According to one embodiment of the invention, the invention combines all process parts in one manageable solution, resulting in a flexible loading scheme based on bucketed incoming data and collected process flow information. The method also detaches the solution from the distinctive contents of a single data flow giving perspective on whole process and resource usage. Furthermore, because data loading and data delivery are separate and independent processes, they can be executed asynchronously.

[0090] Furthermore, the present invention solves the overall challenges, for example, for data warehousing process of 24/7 (or near-realtime) environments. Particularly, the invention solves the problem of staging table (area) handling, which typically faces the challenges of rising high water marks, lack of load throttling based on metrics collected, identifying bad buckets of data, reloading and process part isolation.

[0091] In general, the invention is applicable for all environments that can utilize range or list partitioning on tables. Databases supporting either of these partitioning schemes are e.g. Oracle, MySQL, DB2 v9 (Viper), Teradata and also partly Microsoft SQLServer and Sybase.

[0092] The exemplary embodiments can include, for example, any suitable servers, workstations, and the like, capable of performing the processes of the exemplary embodiments.

[0093] It is to be understood that the exemplary embodiments are for exemplary purposes, as many variations of the specific hardware used to implement the exemplary embodiments are possible, as will be appreciated by those skilled in the hardware and/or software art(s). For example, the functionality of one or more of the components of the exemplary embodiments can be implemented via one or more hardware and/or software devices.

[0094] The exemplary embodiments can store information relating to various processes described herein. This information can be stored in one or more memories, such as a hard disk, optical disk, magnetooptical disk, RAM, and the like. One or more databases can store the information used to implement the exemplary embodiments of the present inventions. The databases can be organized using data structures (e.g., records, tables, arrays, fields, graphs, trees, lists, and the like) included in one or more memories or storage devices listed herein. The processes described with respect to the exemplary embodiments can include appropriate data structures for storing data collected and/or generated by the processes of the devices and subsystems of the exemplary embodiments in one or more databases.

[0095] All or a portion of the exemplary embodiments can be conveniently implemented using one or more general purpose processors, microprocessors, digital signal processors, micro-controllers, and the like, programmed according to the teachings of the exemplary embodiments of the present inventions, as will be appreciated by those skilled in the computer and/or software art(s). Appropriate software can be readily prepared by programmers of ordinary skill based on the teachings of the exemplary embodiments, as will be appreciated by those skilled in the software art. In addition, the exemplary embodiments can be implemented by the preparation of application-specific integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be appreciated by those skilled in the electrical art(s). Thus, the exemplary embodiments are not limited to any specific combination of hardware and/or software.

[0096] Stored on any one or on a combination of computer readable media, the exemplary embodiments of the present inventions can include software for controlling the components of the exemplary embodiments, for driving the components of the exemplary embodiments, for enabling the components of the exemplary embodiments to interact with a human user, and the like. Such software can include, but is not limited to, device drivers, firmware, operating systems, development tools, applications software, and the like. Such computer readable media further can include the computer program product of an embodiment of the present inventions for performing all or a portion (if processing is distributed) of the processing performed in implementing the inventions. Computer code devices of the exemplary embodiments of the present inventions can include any suitable interpretable or executable code mechanism, including but not limited to scripts, interpretable programs, dynamic link libraries (DLLs), Java classes and applets, complete executable programs, Common Object Request Broker Architecture (CORBA) objects, and the like. Moreover, parts of the processing of the exemplary embodiments of the present inventions can be distributed for better performance, reliability, cost, and the like.

[0097] As stated above, the components of the exem-

plary embodiments can include computer readable medium or memories for holding instructions programmed according to the teachings of the present inventions and for holding data structures, tables, records, and/or other data described herein. Computer readable medium can include any suitable medium that participates in providing instructions to a processor for execution. Such a medium can take many forms, including but not limited to, non-volatile media, volatile media, transmission media, and the like. Non-volatile media can include, for example, optical or magnetic disks, magneto-optical disks, and the like. Volatile media can include dynamic memories, and the like. Common forms of computer-readable media can include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, any other suitable magnetic medium, a CD-ROM, CDR, CD-RW, DVD, DVD-ROM, DVD ' RW, DVD ' R, any other suitable optical medium, punch cards, paper tape, optical mark sheets, any other suitable physical medium with patterns of holes or other optically recognizable indicia, a RAM, a PROM, an EPROM, a FLASH-EPROM, any other suitable memory chip or cartridge, a carrier wave or any other suitable medium from which a computer can read.

[0098] While the present inventions have been described in connection with a number of exemplary embodiments, and implementations, the present inventions are not so limited, but rather cover various modifications, and equivalent arrangements, which fall within the purview of prospective claims.

Claims

1. A computer-implemented method for processing data in a data managing system comprising at least one staging area (202) into which data can be loaded,
characterised in that the method comprises:

executing the following source data loading process (206) at least once for a data type of source data:

loading source data from a data source (200) storing source data as rows of data, wherein the loaded amount of source data constitutes a bucket and wherein the bucket comprises at least one row of data;
creating a new partition (210) in a staging area for the bucket;
determining a unique identifier for the bucket, the unique identifier being unique among buckets of the same data type or among all buckets of all data types;
assigning a partition identifier for the partition (210), the partition identifier comprising the unique identifier of the bucket;
tagging each row of data in the bucket with the unique identifier;
storing the bucket in the partition identified by the partition identifier; and
generating bucket metadata for the bucket, the bucket metadata determining properties for the bucket stored in the partition,

executing the following delivery process (208) at least once for a bucket, the delivery process acting as a bucket deliverer having associated metadata:

selecting a bucket stored by the loading process in a partition of a staging area (202);
delivering the selected bucket to a target system (204); and
updating in bucket deliverer's metadata that the delivery process (208) has finished processing the bucket;

executing the following cleaning process at predefined time intervals for a bucket;

checking, from the bucket metadata of a bucket, whether the bucket has been processed by all delivery processes (208) processing the bucket;
dropping the partition containing the bucket, when all delivery processes (208) processing the bucket have processed the bucket; and
updating in the bucket metadata that the bucket has been dropped.

wherein the loading process (206), the delivery process (208) and the cleaning process are executed asynchronously.

2. The method according to claim 1, wherein the bucket metadata comprises at least one of the following:

bucket identifier;
bucket type;
number of records in the bucket;
earliest record in the bucket;
latest record in the bucket;
status of the bucket;
bucket start time;
bucket end time; and
loaded rows per second.

3. The method according to any of claims 1 - 2, further comprising:

storing a duplicate of the bucket into an additional data store during the loading process (206); and
providing the duplicate data with the same

unique identifier.

4. The method according to any of claims 1 - 3, wherein the bucket size is fixed.

5. The method according to any of claims 1 - 3, further comprising:

changing the bucket size at least once.

6. The method according to claim 1, wherein the selecting comprises at least one of the following:

selecting the bucket based on a bucket identifier; and
selecting the bucket based on a time stamp in a bucket metadata.

7. The method according to claim 1, wherein during the execution of the delivery process (208), the method further comprises:

updating status information of the bucket in the bucket metadata to 'delivering', when the delivery process starts delivering the data in the bucket.

8. The method according to any of claims 1 - 7, wherein during the execution of the delivery process (208), the method further comprises:

updating bucket deliverer's metadata, when the delivery process starts processing the bucket, the updating comprising:

adding a deliverer identifier in the bucket deliverer's metadata; and
adding a deliverer start time in the bucket deliver's metadata.

9. The method according to claim 8, wherein during the execution of the delivery process (208), the method further comprises:

updating the bucket deliverer's metadata, when the delivery process ends processing the bucket, the updating comprising:

adding a deliverer end time in the bucket deliverer's metadata.

10. The method according to claim 9, wherein during the execution of the cleaning process, the method further comprises:

checking, whether the bucket deliverer's metadata comprises other deliverer identifiers identifying delivery processes (208) that process the same data type;
checking, whether there exists a deliverer end time for each deliverer identifier; and
updating the status information in the bucket metadata to 'delivered', when there exists a deliverer end time for each deliver identifier.

11. The method according to any of claims 1 - 10, wherein the data management system comprises delivery process metadata for each delivery process (208), the metadata comprising at least one of the following:

an identifier of the delivery process;
a name of the delivery process;
a type of the delivery process;
a description for the delivery process;
priority of the delivery process; and
a target bucket type of the delivery process.

12. A computer program comprising program code configured to perform the method of any of claims 1 - 11 when executed in a data processing device.

13. The computer program according to claim 12, embodied on a computer-readable medium.

14. A data management system (808) comprising at least one staging area (806) in which source data can be stored;
characterised in that the data management system (808) comprises:

a loading module (800) for loading data in the data management system (808), wherein the loading module (800) is configured to execute the following source data loading process at least once for a data type of source data:

load source data from a data source (200)
storing source data as rows of data, wherein the loaded amount of source data constitutes a bucket and wherein the bucket comprises at least one row of data;
create a new partition in a staging area for the bucket;
determine a unique identifier for the bucket, the unique identifier being unique among buckets of the same data type or among all buckets of all data types;
assign a partition identifier for the partition, the partition identifier comprising the unique identifier of the bucket;
tag each row of data in the bucket with the unique identifier;
store the bucket in the partition identified by the partition identifier; and
generate bucket metadata for the bucket, the bucket metadata determining properties

for the bucket stored in the partition;

a delivery module (802) for delivering data in the data management system (808), wherein the delivery module (802) is configured to execute the following delivery process (206) at least once for a bucket , the delivery process acting as a bucket deliverer having associated metadata:

select a bucket stored by the loading process in a partition (210) of a staging area (202, 806);
deliver the selected bucket to a target system (204); and
update in bucket deliverer's metadata that the delivery process (206) has finished processing the bucket; and

a cleaning module (804) for cleaning data in the data management system (808), wherein the cleaning module (804) is configured to execute the following cleaning process at predefined time intervals for a bucket:

checking, from bucket metadata of a bucket, whether the bucket has been processed by all delivery processes (206) processing the bucket;
dropping the partition containing the bucket, when all delivery processes (206) processing the bucket have processed the bucket; and
updating in the bucket metadata that the bucket has been dropped;

wherein the loading module (800), the delivery module (802) and the cleaning module (804) are configured to be executed asynchronously.

15. The data management system (808) according to claim 14, wherein the bucket metadata comprises at least one of the following:

bucket identifier;
bucket type;
number of records in the bucket;
earliest record in the bucket;
latest record in the bucket;
status of the bucket;
bucket start time;
bucket end time; and
loaded rows per second

16. The data management system (808) according to any of claims 14 - 15, wherein the loading module is configured to:

store a duplicate of the bucket into an additional data store during the loading process; and
provide the data with the same unique identifier.

17. The data management system (808) according to any of claims 14 - 15, wherein the bucket size is fixed.

18. The data management system (808) according to any of claims 14 - 17, wherein the loading module is configured to:

change the bucket size at least once.

19. The data management system (808) according to claim 14, wherein the delivery module (802) is configured to select the bucket based on at least of the following:

a bucket identifier; and
a time stamp in a bucket metadata.

20. The data management system (808) according to claim 14, wherein the delivery module (802) is configured to:

update status information of the bucket in the bucket metadata to 'delivering', when starting processing the bucket with the delivery process.

21. The data management system (808) according to any of claims 14 - 20, wherein the delivery module (802) is configured to:

update bucket deliverer's metadata, when the delivery module (802) starts processing the bucket, the updating comprising:

adding a deliverer identifier in the bucket deliverer's metadata; and
adding a deliverer start time in the bucket deliver's metadata.

22. The data management system (808) according to claim 21, wherein the delivery module (802) is configured to:

update the bucket deliverer's metadata, when the delivery module (802) ends processing the bucket, the updating comprising:

adding a deliverer end time in the bucket deliverer's metadata.

23. The data management system (808) according to any of claims 14 - 22, wherein the data management system (808) comprises delivery process metadata for each delivery process (206), the metadata comprising at least one of the following:

an identifier of the delivery process;
a name of the delivery process;
a type of the delivery process;
a description for the delivery process;
priority of the delivery process; and
a target bucket type of the delivery process.

24. The data management system (808) according to any of claims 14 - 23, wherein the cleaning module (804) is configured to:

check, whether a bucket deliverer's metadata comprises other deliverer identifiers identifying delivery processes (206) that process the same data type;
check, whether there exists a deliverer end time for each deliverer identifier; and
update the status information in the bucket metadata to 'delivered', when there exists a deliverer end time for each deliver identifier.

Patentansprüche

1. Computerimplementiertes Verfahren zur Verarbeitung von Daten in einem Datenverwaltungssystem mit mindestens einem Einspeicherungsbereich (202), in welchen Daten geladen werden können, **dadurch gekennzeichnet, dass** das Verfahren umfasst:

mindestens einmaliges Ausführen des folgenden Quelldatenladeprozesses (206) für einen Datentyp von Quelldaten:

Laden von Quelldaten aus einer Datenquelle (200), welche Quelldaten als Zeilen von Daten speichert, wobei die geladene Menge von Quelldaten einen Eimer darstellt, und wobei der Eimer mindestens eine Zeile von Daten umfasst;
Bilden einer neuen Partition (210) in einem Einspeicherungsbereich für den Eimer;
Bestimmen einer eindeutigen Kennung für den Eimer, wobei die eindeutige Kennung unter Eimern des gleichen Datentyps oder unter allen Eimern aller Datentypen eindeutig ist;
Zuordnen einer Partitionskennung für die Partition (210), wobei die Partitionskennung die eindeutige Kennung des Eimers umfasst;
Kennzeichnen jeder Zeile von Daten im Eimer mit der eindeutigen Kennung;
Speichern des Eimers in der Partition, die durch die Partitionskennung identifiziert wird;
Erzeugen von Eimer-Megadaten für den Eimer, wobei die Eimer-Megadaten Eigenschaften für den Eimer bestimmen, der in der Partition gespeichert wird.

mindestens einmaliges Ausführen des folgenden Zustellprozesses (208) für einen Eimer, wobei der Zustellprozess als Eimerzusteller mit assoziierten Metadaten fungiert:

Auswählen eines Eimers, der durch den Ladeprozess in einer Partition des Einspeicherungsbereichs (202) gespeichert wird;
Zustellen des ausgewählten Eimers an ein Zielsystem (204); und
Aktualisieren in Eimerzusteller-Metadaten, dass der Zustellprozess (208) Verarbeitung des Eimers abgeschlossen hat;

Ausführen des folgenden Löschprozesses in vordefinierten Zeitintervallen für einen Eimer:

Prüfen anhand der Eimer-Metadaten eines Eimers, ob der Eimer durch alle Zustellprozesse (208), die den Eimer verarbeiten, verarbeitet wurde;
Verwerfen der Partition, die den Eimer enthält, wenn alle Zustellprozesse (208), die den Eimer verarbeiten, den Eimer verarbeitet haben; und
Aktualisieren in den Eimer-Metadaten, dass der Eimer verworfen wurde,

wobei der Ladeprozess (206), der Zustellprozess (208) und der Löschprozess asynchron ausgeführt werden.

2. Verfahren nach Anspruch 1, wobei die Eimer-Metadaten mindestens eines von Folgendem umfassen:

Eimerkennung;
Eimertyp;
Anzahl von Datensätzen im Eimer;
frühestem Datensatz im Eimer;
letztem Datensatz im Eimer;
Status des Eimers;
Eimer-Startzeit;
Eimer-Endzeit; und
geladenen Zeilen pro Sekunde.

3. Verfahren nach einem der Ansprüche 1 bis 2, ferner umfassend:

Speichern eines Duplikats des Eimers in einem zusätzlichen Datenspeicher während des Ladeprozesses (206); und
Versehen der duplizierten Daten mit der gleichen eindeutigen Kennung.

4. Verfahren nach einem der Ansprüche 1 bis 3, wobei die Eimergröße fest ist.

5. Verfahren nach einem der Ansprüche 1 bis 3, ferner umfassend:

   mindestens einmaliges Ändern der Eimergröße.

6. Verfahren nach Anspruch 1, wobei das Auswählen mindestens eines von Folgendem umfassen:

   Auswählen des Eimers basierend auf einer Eimerkennung; und
   Auswählen des Eimers basierend auf einem Zeitstempel in Eimer-Metadaten.

7. Verfahren nach Anspruch 1, wobei das Verfahren während der Ausführung des Zustellprozesses (208) ferner umfasst:

   Aktualisieren von Statusinformationen des Eimers in den Eimer-Metadaten auf 'Es wird zugestellt', wenn der Zustellprozess mit dem Zustellen der Daten im Eimer beginnt.

8. Verfahren nach einem der Ansprüche 1 bis 7, wobei das Verfahren während der Ausführung des Zustellprozesses (208) ferner umfasst:

   Aktualisieren von Eimerzusteller-Metadaten, wenn der Zustellprozess mit der Verarbeitung des Eimers beginnt, wobei das Aktualisieren umfasst:

      Hinzufügen einer Zustellerkennung in den Eimerzusteller-Metadaten; und
      Hinzufügen einer Zusteller-Startzeit in den Eimerzusteller-Metadaten.

9. Verfahren nach Anspruch 8, wobei das Verfahren während der Ausführung des Zustellprozesses (208) ferner umfasst:

   Aktualisieren der Eimerzusteller-Metadaten, wenn der Zustellprozess die Verarbeitung des Eimers beendet, wobei das Aktualisieren umfasst:

      Hinzufügen einer Zusteller-Endzeit in den Eimerzusteller-Metadaten.

10. Verfahren nach Anspruch 9, wobei das Verfahren während der Ausführung des Löschprozesses ferner umfasst:

    Prüfen, ob die Eimerzusteller-Metadaten andere Zustellerkennungen umfassen, welche Zustellprozesse (208) identifizieren, die den gleichen Datentyp verarbeiten;
    Prüfen, ob eine Zusteller-Endzeit für jede Zustellerkennung vorhanden ist; und
    Aktualisieren der Statusinformationen in den Eimer-Metadaten auf 'zugestellt', wenn eine Zusteller-Endzeit für jede Zustellerkennung vorhanden ist.

11. Verfahren nach einem der Ansprüche 1 bis 10, wobei das Datenverwaltungssystem Zustellprozess-Metadaten für jeden Zustellprozess (208) umfasst, wobei die Metadaten mindestens eines von Folgendem umfassen:

    einer Kennung des Zustellprozesses;
    einem Namen des Zustellprozesses;
    einem Typ des Zustellprozesses;
    einer Beschreibung für den Zustellprozess;
    Priorität des Zustellprozesses; und
    einem Ziel-Eimertyp des Zustellprozesses.

12. Computerprogramm, umfassend Programmcode, der so konfiguriert ist, dass er bei Ausführung in einer Datenverarbeitungsvorrichtung das Verfahren nach einem der Ansprüche 1 bis 11 durchführt.

13. Computerprogramm nach Anspruch 12, das auf einem computerlesbaren Medium enthalten ist.

14. Datenverwaltungssystem (808), umfassend mindestens einen Einspeicherungsbereich (806), in welchem Quelldaten gespeichert werden können; **dadurch gekennzeichnet, dass** das Datenverwaltungssystem (808) umfasst:

    ein Lademodul (800) zum Laden von Daten im Datenverwaltungssystem (808), wobei das Lademodul (800) so konfiguriert ist, dass es den folgenden Quelldatenladeprozess für einen Datentyp von Quelldaten mindestens einmal ausführt:

       Laden von Quelldaten aus einer Datenquelle (200), welche Quelldaten als Zeilen von Daten speichert, wobei die geladene Menge von Quelldaten einen Eimer darstellt, und wobei der Eimer mindestens eine Zeile von Daten umfasst;
       Bilden einer neuen Partition in einem Einspeicherungsbereich für den Eimer;
       Bestimmen einer eindeutigen Kennung für den Eimer, wobei die eindeutige Kennung unter Eimern des gleichen Datentyps oder unter allen Eimern aller Datentypen eindeutig ist;
       Zuordnen einer Partitionskennung für die Partition, wobei die Partitionskennung die eindeutige Kennung des Eimers umfasst:

Kennzeichnen jeder Zeile von Daten im Eimer mit der eindeutigen Kennung;
Speichern des Eimers in der Partition, die durch die Partitionskennung identifiziert wird; und
Erzeugen von Eimer-Megadaten für den Eimer, wobei die Eimer-Megadaten Eigenschaften für den Eimer bestimmen, der in der Partition gespeichert wird;

ein Zustellmodul (802) zum Zustellen von Daten im Datenverwaltungssystem (808), wobei das Zustellmodul (802) so konfiguriert ist, dass es den folgenden Zustellprozess (206) für einen Eimer mindestens einmal ausführt, wobei der Zustellprozess als Eimerzusteller mit assoziierten Metadaten fungiert:

Auswählen eines Eimers, der durch den Ladeprozess in einer Partition (210) des Einspeicherungsbereichs (202, 806) gespeichert wird;
Zustellen des ausgewählten Eimers an ein Zielsystem (204); und
Aktualisieren in den Eimerzusteller-Metadaten, dass der Zustellprozess (206) Verarbeitung des Eimers abgeschlossen hat; und

ein Löschmodul (804) zum Löschen von Daten im Datenverwaltungssystem (808), wobei das Löschmodul (804) so konfiguriert ist, dass es den folgenden Löschprozess für einen Eimer in vordefinierten Zeitintervallen ausführt:

Prüfen anhand von Eimer-Metadaten eines Eimers, ob der Eimer durch alle Zustellprozesse (206), die den Eimer verarbeiten, verarbeitet wurde;
Verwerfen der Partition, die den Eimer enthält, wenn alle Zustellprozesse (206), die den Eimer verarbeiten, den Eimer verarbeitet haben; und
Aktualisieren in den Eimer-Metadaten, dass der Eimer verworfen wurde,

wobei das Lademodul (800), das Zustellmodul (802) und das Löschmodul (804) so konfiguriert sind, dass sie asynchron ausgeführt werden.

15. Datenverwaltungssystem (808) nach Anspruch 14, wobei die Eimer-Metadaten mindestens eines von Folgendem umfassen:

Eimerkennung;
Eimertyp;
Anzahl von Datensätzen im Eimer;
frühestem Datensatz im Eimer;
letztem Datensatz im Eimer;
Status des Eimers;
Eimer-Startzeit;
Eimer-Endzeit; und
geladenen Zeilen pro Sekunde.

16. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 15, wobei das Lademodul konfiguriert ist zum:

Speichern eines Duplikats des Eimers in einem zusätzlichen Datenspeicher während des Ladeprozesses; und
Versehen der Daten mit der gleichen eindeutigen Kennung.

17. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 15, wobei die Eimergröße fest ist.

18. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 17, wobei das Lademodul konfiguriert ist zum:

mindestens einmaligen Ändern der Eimergröße.

19. Datenverwaltungssystem (808) nach Anspruch 14, wobei das Zustellmodul (802) so konfiguriert ist, dass es den Eimer basierend auf mindestens einem von Folgendem auswählt:

einer Eimerkennung; und
einem Zeitstempel in Eimer-Metadaten.

20. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 15, wobei das Zustellmodul (802) konfiguriert ist zum:

Aktualisieren von Statusinformationen des Eimers in den Eimer-Metadaten auf 'Es wird zugestellt' bei Beginn der Verarbeitung des Eimers mit dem Zustellprozess.

21. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 20, wobei das Zustellmodul (802) konfiguriert ist zum:

Aktualisieren von Eimerzusteller-Metadaten, wenn das Zustellmodul (802) mit der Verarbeitung des Eimers beginnt, wobei das Aktualisieren umfasst:

Hinzufügen einer Zustellerkennung in den Eimerzusteller-Metadaten; und
Hinzufügen einer Zusteller-Startzeit in den Eimerzusteller-Metadaten.

22. Datenverwaltungssystem (808) nach Anspruch 21,

wobei das Zustellmodul (802) konfiguriert ist zum:

Aktualisieren der Eimerzusteller-Metadaten, wenn das Zustellmodul (802) die Verarbeitung des Eimers beendet, wobei das Aktualisieren umfasst:

Hinzufügen einer Zusteller-Endzeit in den Eimerzusteller-Metadaten.

23. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 22, wobei das Datenverwaltungssystem (808) Zustellprozess-Metadaten für jeden Zustellprozess (206) umfasst, wobei die Metadaten mindestens eines von Folgendem umfassen:

einer Kennung des Zustellprozesses;
einem Namen des Zustellprozesses;
einem Typ des Zustellprozesses;
einer Beschreibung für den Zustellprozess;
Priorität des Zustellprozesses; und
einem Ziel-Eimertyp des Zustellprozesses.

24. Datenverwaltungssystem (808) nach einem der Ansprüche 14 bis 23, wobei das Löschmodul (804) konfiguriert ist zum:

Prüfen, ob Eimerzusteller-Metadaten andere Zustellerkennungen umfassen, welche Zustellprozesse (206) identifizieren, die den gleichen Datentyp verarbeiten;
Prüfen, ob eine Zusteller-Endzeit für jede Zustellerkennung vorhanden ist; und
Aktualisieren der Statusinformationen in den Eimer-Metadaten auf 'zugestellt', wenn eine Zusteller-Endzeit für jede Zustellerkennung vorhanden ist.

Revendications

1. Procédé mis en oeuvre par ordinateur, pour le traitement de données dans un système de gestion de données comprenant au moins une zone de préparation (202) dans laquelle des données peuvent être chargées,
caractérisé en ce que le procédé consiste à :

exécuter au moins une fois le traitement de chargement de données sources suivant (206) pour un type de données de données sources, consistant à :

charger des données sources d'une source de données (200) contenant en mémoire des données sources sous forme de rangées de données, dans lequel la quantité chargée de données sources constitue un groupement et dans lequel le groupement comprend au moins une rangée de données ;
créer une nouvelle partition (210) destinée au groupement dans une zone de préparation ;
déterminer un identificateur unique du groupement, l'identificateur unique étant unique parmi des groupements du même type de données ou parmi tous les groupements de tous les types de données ;
attribuer un identificateur de partition à la partition (210), l'identificateur de partition comprenant l'identificateur unique du groupement ;
étiqueter chaque rangée de données du groupement avec l'identificateur unique ;
mémoriser le groupement dans la partition identifiée par l'identificateur de partition ; et
générer des métadonnées de groupement du groupement, les métadonnées de groupement déterminant des propriétés du groupement mémorisé dans la partition,

exécuter au moins une fois le traitement de distribution suivant (208) pour un groupement, le traitement de distribution servant de distributeur de groupement comportant des métadonnées associées, consistant à :

sélectionner un groupement mémorisé par le traitement de chargement dans une partition d'une zone de préparation (202) ;
distribuer le groupement sélectionné à un système cible (204) ; et
mettre à jour des métadonnées du distributeur de groupement indiquant que le traitement de distribution (208) a terminé de traiter le groupement ;

exécuter le traitement de nettoyage suivant d'un groupement à des intervalles de temps prédéfinis, consistant à :

vérifier, à partir des métadonnées de groupement d'un groupement, si le groupement a fait l'objet de tous les traitements de distribution (208) de traitement du groupement ;
éliminer la partition contenant le groupement, lorsque tous les traitements de distribution (208) de traitement du groupement ont traité le groupement ; et
mettre à jour les métadonnées de groupement indiquant que le groupement a été éliminé,

dans lequel le traitement de chargement (206),

le traitement de distribution (208) et le traitement de nettoyage sont exécutés de façon asynchrone.

2. Procédé selon la revendication 1, dans lequel les métadonnées de groupement comprennent au moins l'une des composantes suivantes :

un identificateur de groupement ;
un type de groupement ;
un nombre d'enregistrements dans le groupement ;
l'enregistrement le plus ancien dans le groupement ;
l'enregistrement le plus récent dans le groupement ;
un état du groupement ;
un instant de début de groupement ;
un instant de fin de groupement ; et
les rangées chargées par seconde.

3. Procédé selon l'une quelconque des revendications 1 et 2, consistant en outre à :

mémoriser une reproduction du groupement dans une mémoire de données supplémentaire pendant le traitement de chargement (206) ; et attribuer le même identificateur unique aux données reproduites.

4. Procédé selon l'une quelconque des revendications 1 à 3, dans lequel le groupement a une taille fixe.

5. Procédé selon l'une quelconque des revendications 1 à 3, consistant en outre à :

modifier au moins une fois la taille de groupement.

6. Procédé selon la revendication 1, dans lequel la sélection comprend au moins l'une des sélections suivantes :

une sélection du groupement sur la base d'un identificateur de groupement ; et
une sélection de groupement sur la base d'une estampille temporelle de métadonnées de groupement.

7. Procédé selon la revendication 1, dans lequel, pendant l'exécution du traitement de distribution (208), le procédé consiste en outre à :

mettre à jour des informations d'état du groupement des métadonnées de groupement à l'état « en cours de distribution », lorsque le traitement de distribution commence la distribution des données du groupement.

8. Procédé selon l'une quelconque des revendications 1 à 7, dans lequel, pendant l'exécution du traitement de distribution (208), le procédé consiste en outre à :

mettre à jour des métadonnées du distributeur de groupement, lorsque le traitement de distribution commence à traiter le groupement, la mise à jour consistant à :

ajouter un identificateur de distributeur aux métadonnées du distributeur de groupement ; et
ajouter un instant de début du distributeur aux métadonnées du distributeur de groupement.

9. Procédé selon la revendication 8, dans lequel, pendant l'exécution du traitement de distribution (208), le procédé consiste en outre à :

mettre à jour les métadonnées du distributeur de groupement, lorsque le traitement de distribution termine le traitement du groupement, la mise à jour consistant à :

ajouter un instant de fin de distributeur aux métadonnées du distributeur de groupement.

10. Procédé selon la revendication 9, dans lequel, pendant l'exécution du traitement de nettoyage, le procédé consiste en outre à :

vérifier si les métadonnées de distributeur de groupement comprennent d'autres identificateurs de distributeur identifiant des traitements de distribution (208) qui traitent le même type de données ;
vérifier la présence éventuelle d'un instant de fin de distributeur de chaque identificateur de distributeur ; et
mettre à jour les informations d'état des métadonnées de groupement à l'état « distribué », lors de la présence d'un instant de fin de distributeur pour chaque identificateur de distribution.

11. Procédé selon l'une quelconque des revendications 1 à 10, dans lequel le système de gestion de données comprend des métadonnées de traitement de distribution pour chaque traitement de distribution (208), les métadonnées comprenant au moins une composante parmi les composantes suivantes :

un identificateur du traitement de distribution ;
un nom du traitement de distribution ;
un type du traitement de distribution ;
une description du traitement de distribution ;

une priorité attribuée au traitement de distribution ; et
un type de groupement cible du traitement de distribution.

12. Programme informatique comprenant un code de programme conçu pour mettre en oeuvre le procédé selon l'une quelconque des revendications 1 à 11 lorsqu'il est exécuté dans un dispositif de traitement de données.

13. Programme informatique selon la revendication 12, intégré à un support lisible par ordinateur.

14. Système de gestion de données (808) comprenant au moins une zone de préparation (806) dans laquelle des données sources peuvent être mémorisées ;
caractérisé en ce que le système de gestion de données (808) comprend :

un module de chargement (800) servant à charger des données dans le système de gestion de données (808), dans lequel le module de chargement (800) est configuré pour exécuter au moins une fois le traitement de chargement de données sources suivant pour un type de données de données sources, consistant à :

charger des données sources d'une source de données (200) contenant en mémoire des données sources sous forme de rangées de données, dans lequel la quantité chargée de données sources constitue un groupement et dans lequel le groupement comprend au moins une rangée de données ;
créer une nouvelle partition destinée au groupement dans une zone de préparation ;
déterminer un identificateur unique du groupement, l'identificateur unique étant unique parmi des groupements du même type de données ou parmi tous les groupements de tous les types de données ;
attribuer un identificateur de partition à la partition, l'identificateur de partition comprenant l'identificateur unique du groupement ;
étiqueter chaque rangée de données du groupement avec l'identificateur unique ;
mémoriser le groupement dans la partition identifiée par l'identificateur de partition ; et
générer des métadonnées de groupement du groupement, les métadonnées de groupement déterminant des propriétés du groupement mémorisé dans la partition ;

un module de distribution (802) servant à distribuer des données dans le système de gestion de données, dans lequel le module de distribution (802) est configuré pour exécuter au moins une fois le traitement de distribution suivant (206) pour un groupement, le traitement de distribution servant de distributeur de groupement comportant des métadonnées associées, consistant à :

sélectionner un groupement mémorisé par le traitement de chargement dans une partition (210) d'une zone de préparation (202, 806) ;
distribuer le groupement sélectionné à un système cible (204) ; et
mettre à jour des métadonnées du distributeur de groupement indiquant que le traitement de distribution (206) a terminé de traiter le groupement ; et

un module de nettoyage (804) servant à nettoyer des données dans le système de gestion de données (808), dans lequel le module de nettoyage (804) est configuré pour exécuter le traitement de nettoyage suivant à des intervalles de temps prédéfinis pour un groupement, consistant à :

vérifier, à partir de métadonnées de groupement d'un groupement, si le groupement a été traité par tous les traitements de distribution (206) de traitement du groupement ;
éliminer la partition contenant le groupement, lorsque tous les traitements de distribution (206) de traitement du groupement ont traité le groupement ; et
mettre à jour les métadonnées de groupement indiquant que le groupement a été éliminé ;

dans lequel le module de chargement (800), le module de distribution (802) et le module de nettoyage (804) sont configurés pour une exécution asynchrone.

15. Système de gestion de données (808) selon la revendication 14, dans lequel les métadonnées de groupement comprennent au moins l'une des composantes suivantes :

un identificateur de groupement ;
un type de groupement ;
un nombre d'enregistrements dans le groupement ;
l'enregistrement le plus ancien dans le groupement ;
l'enregistrement le plus récent dans le

groupement ;
un état du groupement ;
un instant de début de groupement ;
un instant de fin de groupement ; et
les rangées chargées par seconde.

16. Système de gestion de données (808) selon l'une quelconque des revendications 14 et 15, dans lequel le module de chargement est configuré pour :

mémoriser une reproduction du groupement dans une mémoire de données supplémentaire pendant le traitement de chargement ; et
attribuer le même identificateur unique aux données.

17. Système de gestion de données (808) selon l'une quelconque des revendications 14 et 15, dans lequel le groupement a une taille fixe.

18. Système de gestion de données (808) selon l'une quelconque des revendications 14 à 17, dans lequel le module de chargement est configuré pour :

modifier au moins une fois la taille de groupement.

19. Système de gestion de données (808) selon la revendication 14, dans lequel le module de distribution (802) est configuré pour sélectionner le groupement sur la base d'au moins l'une des composantes suivantes :

un identificateur de groupement ; et
une estampille temporelle des métadonnées de groupement.

20. Système de gestion de données (808) selon la revendication 14, dans lequel le module de distribution (802) est configuré pour :

mettre à jour des informations d'état du groupement des métadonnées de groupement à l'état « en cours de distribution », au début du traitement du groupement au moyen du traitement de distribution.

21. Système de gestion de données (808) selon l'une quelconque des revendications 14 à 20, dans lequel le module de distribution (802) est configuré pour :

mettre à jour des métadonnées du distributeur de groupement, lorsque le module de distribution (802) commence à traiter le groupement, la mise à jour consistant à :

ajouter un identificateur de distributeur aux métadonnées du distributeur de groupement ; et
ajouter un instant de début de distributeur aux métadonnées du distributeur de groupement.

22. Système de gestion de données (808) selon la revendication 21, dans lequel le module de distribution (802) est configuré pour :

mettre à jour les métadonnées du distributeur de groupement, lorsque le module de distribution (802) termine le traitement du groupement, la mise à jour consistant à :

ajouter un instant de fin de distributeur aux métadonnées du distributeur de groupement.

23. Système de gestion de données (808) selon l'une quelconque des revendications 14 à 22, dans lequel le système de gestion de données (808) comprend des métadonnées de traitement de distribution pour chaque traitement de distribution (206), les métadonnées comprenant au moins l'une des composantes suivantes :

un identificateur du traitement de distribution ;
un nom du traitement de distribution ;
un type du traitement de distribution ;
une description du traitement de distribution ;
une priorité attribuée au traitement de distribution ; et
un type de groupement cible du traitement de distribution.

24. Système de gestion de données (808) selon l'une quelconque des revendications 14 à 23, dans lequel le module de nettoyage (804) est configuré pour :

vérifier si des métadonnées du distributeur de groupement comprennent d'autres identificateurs de distributeur identifiant des traitements de distribution (206) qui traitent le même type de données ;
vérifier la présence éventuelle d'un instant de fin de distributeur pour chaque identificateur de distributeur ; et
mettre à jour les informations d'état des métadonnées de groupement à l'état « distribué », lors de la présence d'un instant de fin de distributeur pour chaque identificateur de distributeur.

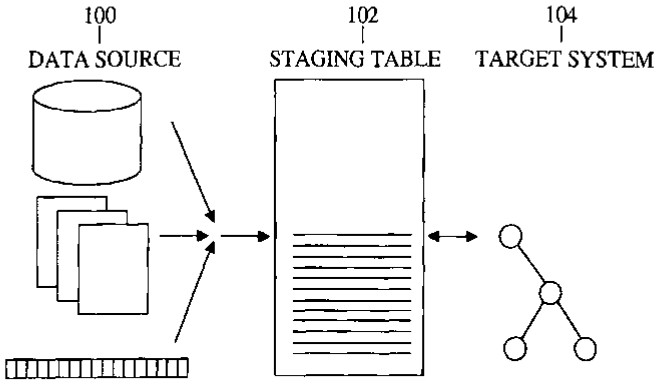

FIG. 1

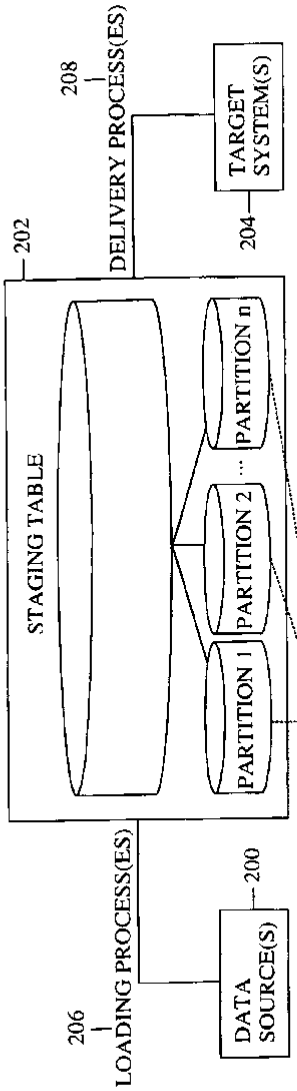
208
DELIVERY PROCESS(ES)
TARGET SYSTEM(S)
204
202
STAGING TABLE
PARTITION n
PARTITION 2
PARTITION 1
210
206
LOADING PROCESS(ES)
DATA SOURCE(S)
200

FIG. 2

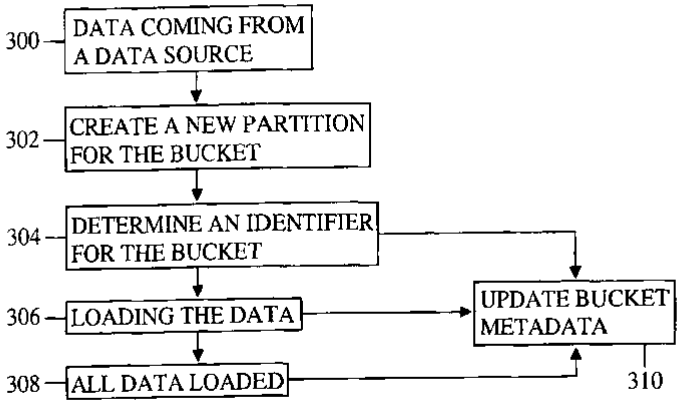

FIG. 3A

CTRL_BUCKET		320
BUCKET_ID	NUMBER(10)	326
BUCKET_TYPE	VARCHAR2(20)	328
NUMBER_OF_ROWS	NUMBER(6)	330
EARLIEST_RECORD	DATETIME	332
LATEST_RECORD	DATETIME	334
STATUS	VARCHAR2(20)	336
BUCKET_STARTTIME	DATETIME	338
BUCKET_ENDTIME	DATETIME	340
ROWS_PER_SEC	NUMBER(6)	342
322	324	

FIG. 3B

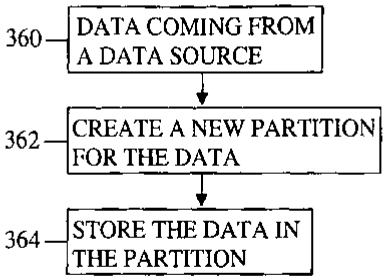

FIG. 3C

CTRL_DELIVERER		400
DELIVERER_ID	NUMBER(4)	402
DELIVERER_NAME	VARCHAR2(20)	404
DELIVERER_TYPE	VARCHAR2(10)	406
DELIVERER_DESCRIPTION	VARCHAR2(100)	408
PRIORITY	NUMBER(2)	410
DELIVERER_TARGET	VARCHAR2(20)	412

414 416

FIG. 4

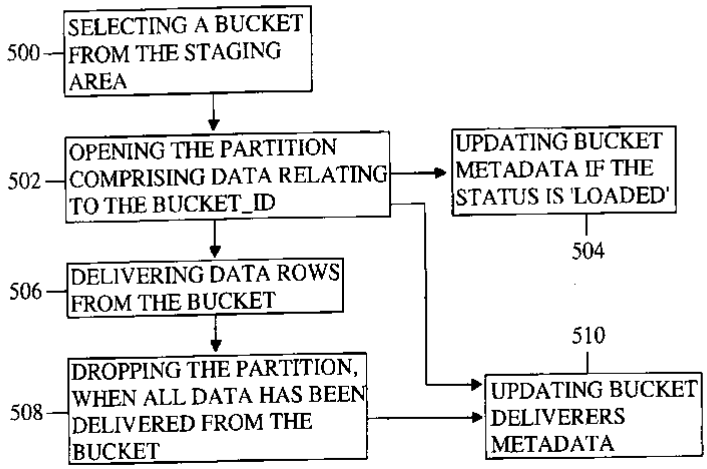

FIG. 5A

CTRL_BUCKET_DELIVERERS		520
BUCKET_ID	NUMBER(10)	522
DELIVERER_ID	NUMBER(4)	524
DELIVERER_STARTTIME	DATETIME	526
DELIVERER_ENDTIME	DATETIME	528
ROWS_PER_SEC	NUMBER(6)	530

532 534

FIG. 5B

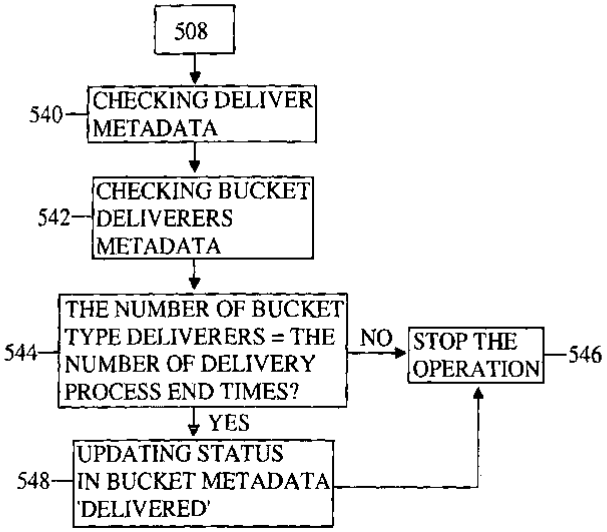
508
540
CHECKING DELIVER METADATA
542
CHECKING BUCKET DELIVERERS METADATA
544
THE NUMBER OF BUCKET TYPE DELIVERERS = THE NUMBER OF DELIVERY PROCESS END TIMES?
NO
STOP THE OPERATION
546
YES
548
UPDATING STATUS IN BUCKET METADATA 'DELIVERED'

FIG. 5C

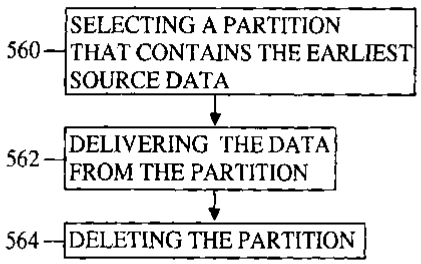
560
SELECTING A PARTITION THAT CONTAINS THE EARLIEST SOURCE DATA
562
DELIVERING THE DATA FROM THE PARTITION
564
DELETING THE PARTITION

FIG. 5D

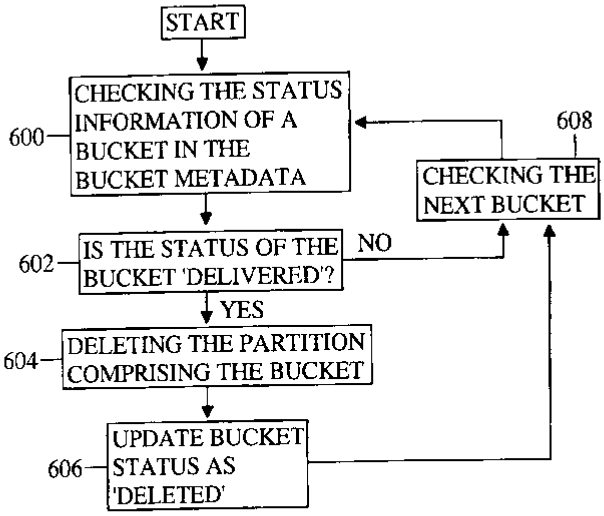
START
CHECKING THE STATUS INFORMATION OF A BUCKET IN THE BUCKET METADATA
600
IS THE STATUS OF THE BUCKET 'DELIVERED'?
602
NO
YES
DELETING THE PARTITION COMPRISING THE BUCKET
604
UPDATE BUCKET STATUS AS 'DELETED'
606
CHECKING THE NEXT BUCKET
608

FIG. 6A

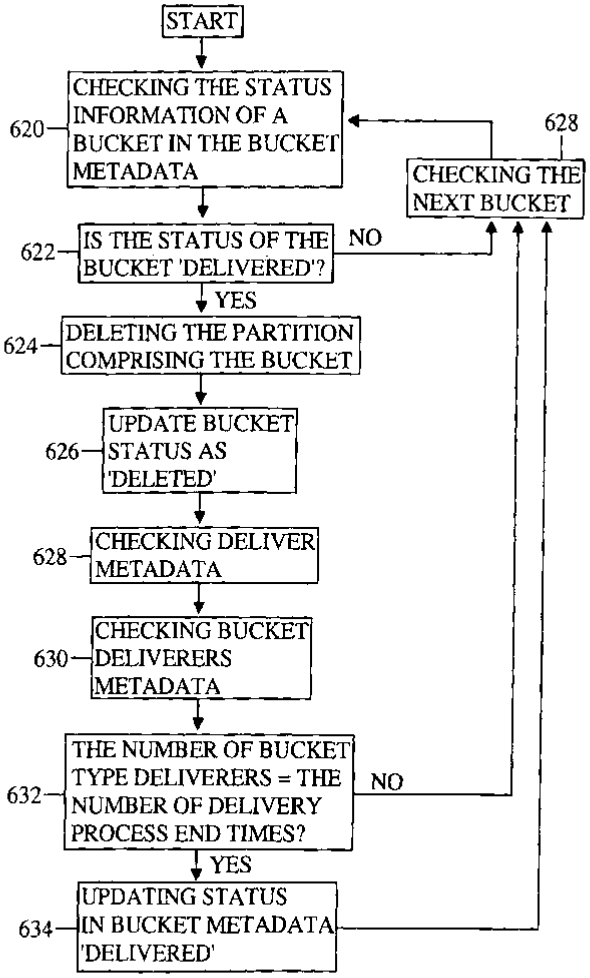
START
CHECKING THE STATUS INFORMATION OF A BUCKET IN THE BUCKET METADATA
620
IS THE STATUS OF THE BUCKET 'DELIVERED'?
622
NO
YES
CHECKING THE NEXT BUCKET
628
DELETING THE PARTITION COMPRISING THE BUCKET
624
UPDATE BUCKET STATUS AS 'DELETED'
626
CHECKING DELIVER METADATA
628
CHECKING BUCKET DELIVERERS METADATA
630
THE NUMBER OF BUCKET TYPE DELIVERERS = THE NUMBER OF DELIVERY PROCESS END TIMES?
632
NO
YES
UPDATING STATUS IN BUCKET METADATA 'DELIVERED'
634

FIG. 6B

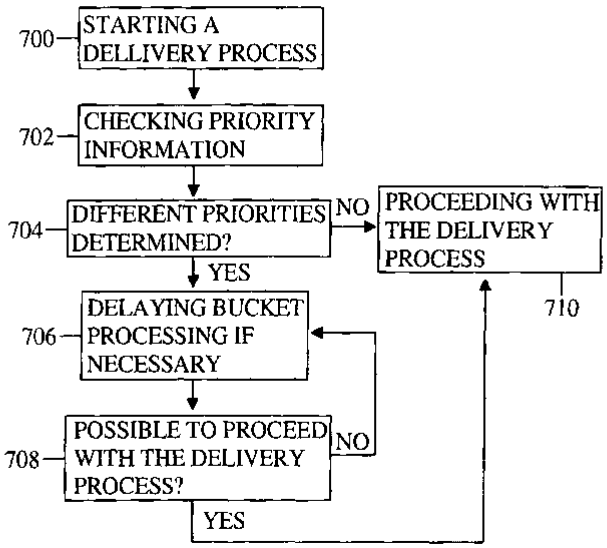
700
STARTING A DELLIVERY PROCESS
702
CHECKING PRIORITY INFORMATION
704
DIFFERENT PRIORITIES DETERMINED?
NO
PROCEEDING WITH THE DELIVERY PROCESS
710
YES
706
DELAYING BUCKET PROCESSING IF NECESSARY
708
POSSIBLE TO PROCEED WITH THE DELIVERY PROCESS?
NO
YES

FIG. 7

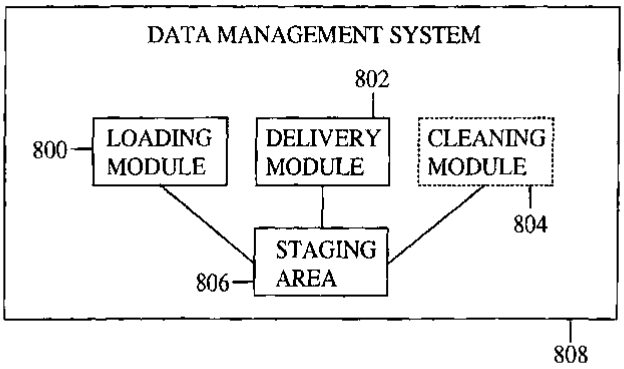
DATA MANAGEMENT SYSTEM
802
800
LOADING MODULE
DELIVERY MODULE
CLEANING MODULE
804
STAGING AREA
806
808

FIG. 8

REFERENCES CITED IN THE DESCRIPTION

This list of references cited by the applicant is for the reader's convenience only. It does not form part of the European patent document. Even though great care has been taken in compiling the references, errors or omissions cannot be excluded and the EPO disclaims all liability in this regard.

Patent documents cited in the description

- EP 1591914 A **[0007]**
- US 7093232 B1 **[0008]**
- US 2004225664 A **[0009]**

Non-patent literature cited in the description

- **A. EJAZ et al.** Utilizing Staging Tables in Data Integration to Load Data into Materialized Views. Springer Verlag, 2004, vol. 3314/2004, 685-691 **[0010]**