Online Expansion of Large-scale Data Warehouses

Jeffrey Cohen† John Eshleman Brian Hagenbuch† Joy Kent†
Christopher Pedrotti† Gavin Sherry† Florian Waas†

†EMC Corp.
Data Computing Division
firstname.lastname@emc.com

ABSTRACT

Modern data warehouses store exceedingly large amounts of data, generally considered the crown jewels of an enterprise. The amount of data maintained in such data warehouses increases significantly over time—often at a continuous pace, e.g., by gathering additional data or retaining data for longer periods to derive additional business value, but occasionally also precipitously, e.g., when consolidating disparate data warehouses and Data Marts into a single database. Having to *expand* a data warehouse with 100's of TB of data by a substantial portion, e.g., 100% or more is a complex and disruptive maintenance operation as it typically involves some sort of dumping and reloading of data which requires substantial downtime.

In this paper we describe the methodology and mechanisms we developed in Greenplum Database to expand large-scale data warehouses in an *online* fashion, i.e., without noticeable downtime. At the core of our approach is a set of robust and transactionally consistent primitives that enable efficient data movement. Special emphasis was put on usability and control that lets an administrator tailor the expansion process to specific operational characteristics via priorities and schedules.

We present a number of experiments to quantify the impact of an on-going expansion on query workloads.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*online expansion, administration, performance*

1. INTRODUCTION

Petabyte-scale data warehousing has pushed the envelope of conventional database technology substantially in the last couple of years. Massively parallel processing of increasingly larger data sets has enabled unprecedented access to data and redefined the role of analytics and transformed it from being considered optional to mission critical. Today, data analytics is one of the fastest growing sectors in the data management business.

Besides classic sales data, modern data warehouses store and aggregate numerous data sources that describe user behavior, e.g., click stream data in Internet applications, sensor data, or call-data-records in telephony to name but a few. And although storage capacity has been increasing rapidly, data keeps out-pacing hardware developments. In particular, data sets grow primarily along two dimensions:

Additional data sources. Analytics in large enterprises is often limited to select, carefully prioritized sources. Often, businesses strive to replicate the success of initial analytics projects and include additional data sources into their data warehouses such as additional applications, improved or more detailed monitoring capabilities etc.

Longer data retention periods. Data warehouses store usually only a relatively short window of data such as the last 90 days. For a number of applications this may be already a useful range. However, in almost all application scenarios longer retention of data results directly in higher quality analytics and increased usefulness and monetization. In application areas like fraud detection, extremely long data retention periods are desirable, including the option of infinite retention.

Another critical dimension of size for a data warehouse is the computing capability it provides, i.e., CPU and memory resources available for query processing. Like with data capacity, increased demand stems from new analytics applications being developed. Providing additional capacity be it storage, CPU or memory is a hard operational challenge especially when business demands call for expansion of an existing system by significant increments, e.g., 100% or more. Over-provisioning is not a viable option, in general. Therefore, it is highly desirable that a data warehouse solution provides *expansion* as a first-class operation.

In this paper we discuss the design and development of an expansion strategy for Greenplum Database, a large-scale MPP database. The central value proposition of large-scale MPP data warehouses, built using commodity shared-nothing systems, is the componentization of individual units without central bottlenecks or significant dependencies between systems such as shared disks etc. Naturally, expanding a shared-nothing system should be as simple as adding additional shared-nothing hosts to the system. From an operational point of view, however, there are a number of hard

Proceedings of the VLDB Endowment, Vol. 4, No. 12

problems to be solved in order to expand a large cluster. Primarily, the biggest challenge is to integrate a set of additional hosts into a system with (1) basically no downtime and (2) manageable and transparent performance characteristics. All while providing the levels of fault-tolerance users have come to expect.

Our solution is based on simple and transactionally consistent primitives that allow us to layer the expansion process. We implemented a number of control tables to empower administrators to control all aspects of expansion, specifically scheduling and monitoring of the on-going expansion work.

Related Work. Elastic expansion of server farms has been explored in a variety of settings including hosted environments [1]. In the case of state-less applications expansion is a rather straight-forward operation as only very small amounts of data need to be moved. The techniques developed in that field are not suitable for a distributed database system, however. Moving for example 10 TB of data requires significant amounts of resources and time. Stateless cluster expansion does not take this into account and provides no mechanisms to coordinate an expansion.

Distributed Hash Tables are data structures specifically designed for flexible group membership of individual storage hosts [12]. They distribute/redistribute data on-the-fly and allow adding and removing hosts rather seamlessly. These techniques do not provide actual database functionality. To the contrary, much of the flexibility can be achieved because they do *not* have to provide transactional consistency. Also, they are designed to expand systems by small increments only and may not be able to keep up with the demand for new resources. Similarly, systems built atop of Distributed Hash Tables like Amazon's Dynamo [5] or Apache's Cassandra [8] inherit these properties.

Hadoop, Apache's version of MapReduce [3] which has been successfully used in a series of data analysis application scenarios [10] provides mechanisms for expansion. Unfortunately, Hadoop's fault-tolerance model is at odds with transactional consistency and the mechanisms developed in this area cannot be transferred directly to database technology.

In the database field online reorganization has a long-standing tradition [11]. To some degree, most of the principles, e.g., index reorganization etc., are applicable to MPP databases as well yet orthogonal to the problem at hand in that they deal with reorganizing of data structures within a stand-alone database—not with the redistribution of data across several servers.

To the best of our knowledge there is no published work on database systems that are able to redistribute large amount of data and provide full functionality at the same time.

Roadmap. The remainder of this paper is organized as follows: in Section 2 we explore the requirements such a feature has to address in a production system from various angles. To illustrate the technical framework to which our work applies, we briefly survey the components of Greenplum Database that are relevant for the understanding of the feature in Section 3. In Section 4 we describe the different layers and technical implements we have built. In the subsequent sections we describe performance experiments that illustrate the operational characteristics of the resulting system and describe our initial experience in the field.

2. DESIDERATA

For an expansion strategy to be successful, a number of requirements must be met.

1. *Scalability of capacity.* The capacity of the expanded system should be in line with default capacity planning, i.e., expanding a system by X% should provide the same capacity as if the system had originally been implemented as 100+X%. In other words, the added capacity is used in the same way as the original.

2. *Scalability of performance.* Analogous to 1, the performance of the expanded system must equal that of a system that was built-out originally at the same size.

3. *Uninterrupted service.* Regular workloads, both scheduled and ad-hoc, must not be interrupted. A short scheduled downtime period such as required for restarting the system, may be acceptable though. In particular, the downtime must be independent of the size of the system before and after expansion as well as independent of the size of the data currently stored in the system, i.e., any potential redistribution of data must not require downtime.

4. *Fault-tolerance.* During expansion standard fault-tolerance mechanisms must not be suspended. A system that provides *k-safety*—up to k components may fail without impairing the system's up time—must be able to tolerate k failures including both old and new components during expansion.

5. *Replication and Disaster Recovery.* Analogous to requirements pertaining to fault-tolerance, any replication mechanisms must continue to function during expansion. This holds for replication as well as restore mechanisms needed in case of a failure or a catastrophic event.

6. *Transparency of process.* Large-scale data warehouses employ highly complex software and hardware components. In order to provide certain service level agreements, administration of the system must be sufficiently transparent, i.e., administrators must be able to reconstruct, to a certain degree, the internal workings of the system. The process of expansion must be easy to diagnose and troubleshoot.

7. *Configurable process.* An expansion may be a long-running process, e.g., in order to satisfy 1 and 6, the expanded system must achieve a certain symmetry with regards to data placement. Depending on the time needed, the expansion process must be fit into a schedule of ongoing operations.

   - Ability to pause/resume; both in an ad-hoc fashion and according to a predetermined schedule;
   - Prioritize data sets; if not all data sets can be expanded without temporarily degraded performance enable administrators and business users to express their preferences regarding their working sets;

8. *Support Data Warehousing specific data patterns.* A significant ratio of data in a data warehouse is stored

in *fact tables*. Fact tables are extended frequently, and, once loaded, queried in read-only fashion. In addition, fact tables are partitioned heavily for operational reasons, in particular, to facilitate loading and rolling off of data. In the vast majority of cases fact tables are partitioned on a per-day basis.

9. *Leverage existing infrastructure.* Ideally, the expansion mechanism re-uses components so as to avoid increasing the complexity of the product.

Not all of these are *hard* requirements, i.e., an expansion mechanism may not meet it but must sufficiently mitigate the resulting effects.

Similarly, not all of the above are equally important: during our requirements analysis we met with a variety of customers and their users and administrators. It quickly became clear that expansion of a large-scale data warehouse is much more intricate than a simple requirement of "no downtime". Specifically, transparency and providing users with a maximum degree of control over the process are mandatory—if hard to quantify.

3. ARCHITECTURE OVERVIEW

The expansion mechanisms we present in this paper are rather general and are applicable to a wide variety of database systems based on MPP shared-nothing architectures. This section surveys the basic principles and architectural considerations that went into building Greenplum Database [13, 2].

Greenplum Database, as shown in Figure 1, is an MPP shared-nothing architecture built from commodity hardware components, i.e., no proprietary hardware is used. Moreover, one of the underlying design principles was to avoid customizing the software explicitly to take advantage of specific hardware properties—rather, the software abstracts the platform. Not only does this result in better portability between different operating systems—Greenplum Database supports several different Linux, Solaris, and for development purposes Mac OS versions—but makes the system less sensitive to variations in hardware configurations.

3.1 Basics

The system distinguishes two types of hosts: (1) a *master host* and (2) *segment hosts*. The master accepts incoming connections and after optimizing a statement or query sends a parallel query plan to the segment databases to do the processing. Each segment host holds one or more *segment databases*. If results need to be returned to the client—as is the case in query processing—they are gathered on the master and forwarded to the client.

Greenplum Database manages two types of segment databases: *primaries* and *mirrors*. A mirror is a logical copy of the primary. Mirrors and primaries are placed across the segment hosts in configurable patterns. By default each segment host will hold N primaries and N mirrors. Mirrors are used in read queries only after a primary is down, in which case they are upgraded to become the acting primary. Otherwise, mirrors simply replicate write activity from the primary in a synchronous fashion. As one should expect with a database system, all data management is transactionally consistent.

3.2 Data Distribution

Besides catalog tables, which are located on the master only, all data is distributed across the segments. Greenplum Database offers several modes of assigning data to segments.

The most prominent is distribution by hashing of the designated *distribution columns* of each table. The concept of using one or more columns to determine the distribution of data provides users with the ability to align tables to improve the processing of frequently encountered or particularly important query patterns. In order to designate a set of columns as distribution columns, a syntax extension is used. In this example, the data of the `nation` table of the TPC-H Benchmark schema will be distributed based on the hash function over the column `n_nationkey`.

```
CREATE TABLE nation
   (
    n_nationkey INTEGER NOT NULL,
    n_name CHARACTER(25) NOT NULL,
    n_regionkey INTEGER NOT NULL,
    n_comment CHARACTER VARYING
   )
DISTRIBUTED BY (n_nationkey)
```

In addition to hash partitioning of data, Greenplum Database also provides a special type of distribution labeled `RANDOMLY`. This type of distribution is particularly useful if a table has only a very small number of distinct rows. In this case, hashing would assign the data to a small number of segments only. In contrast, distributing the data randomly avoids this problem by assigning data based on a round-robin schema to segments. In the following sections we will make extensive use of various distribution types.

In addition, Greenplum Database uses several transient distribution types (see below) during execution of a query; these distributions are mentioned here only for completeness; they are not used for materializing data on disk.

3.3 Query Processing

Based on the data they access, we distinguish three categories of queries:

- *Master-only queries.* These queries involve only data located on the master, e.g., catalog tables, or expressions that can be evaluated on the master without dispatching the query plan to the segment databases.

- *Symmetrically dispatched queries.* All segments execute a symmetric query plan, i.e., each segment executes the same set of operators although over different sets of data. A query plan may contain operators that distribute data between segments including the case of concentrating all data in a single segment as well as redistributing all data from a single segment to all other segments. This is the standard situation for almost all non-trivial queries over user data. See also e.g., [7] for a broader discussion of this technique.

- *Targeted-dispatch queries.* In this case, all data relevant for the query is located on a single segment database; the query plan is dispatched only to a single segment database instead to all segment databases. A typical example of this type of query are singleton lookups.

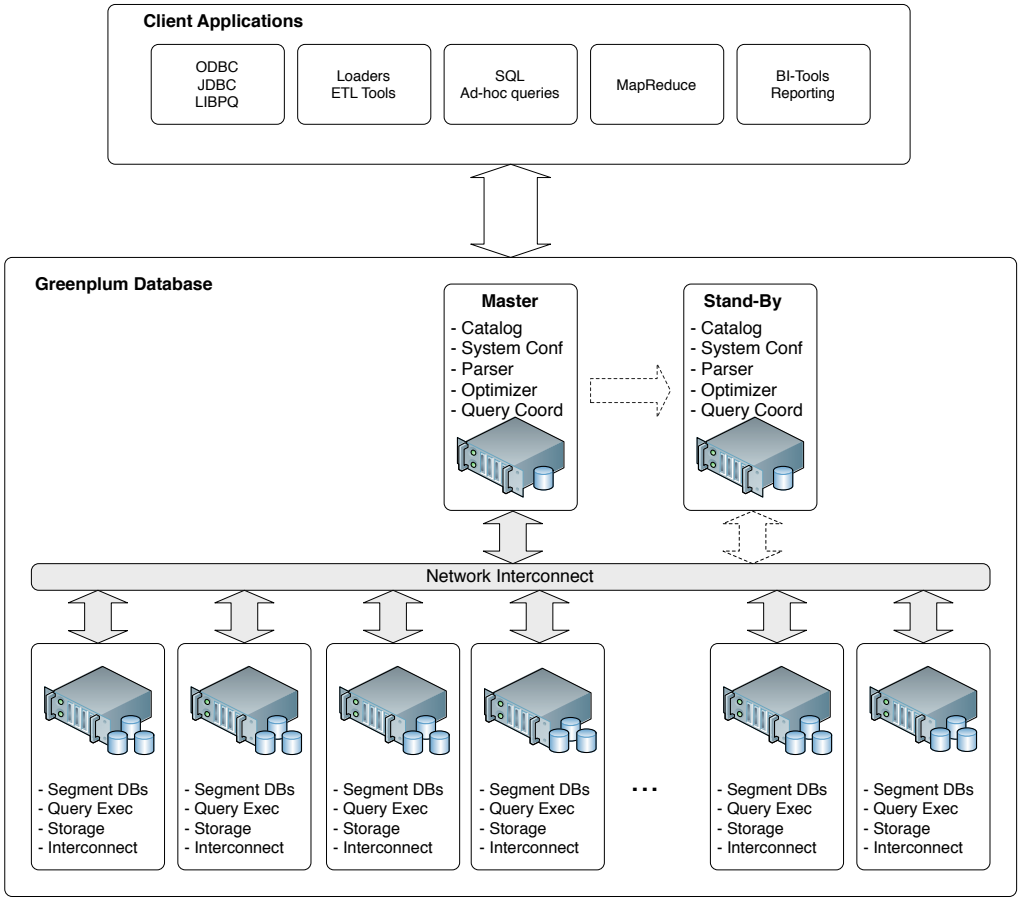


Figure 1: Architecture of Greenplum Database.

This categorization of queries is closely related to the distributions of tables. The distribution of each table is recorded in the catalog and known to the query optimizer at compile time. Based on this knowledge, the optimizer compiles a query plan that takes into account where the data is located. Operations like join or aggregation require specific data distribution in order to guarantee correct results. We discuss this here in more depth because it will be one of the key ingredients for the expansion mechanism we present in Section 4. The optimizer creates a plan that either correctly redistributes the data as needed and exploits the pre-existing distribution or co-location of data whenever possible. To this end, the query processor uses different types of data redistribution operators:

- *Redistribute N:N.* The input relation is redistributed according to the values of a set of columns. This type of distribution operator is typically used when redistributing relations to co-locate them for equi-joins or for grouping of rows.

- *Gather N:1.* All data is sent to one single segment database. This type of distribution operator is usually placed at the root of the plan to concentrate all data in the master segment; it may also be used in the middle of a plan before executing an operation that cannot be executed in parallel on different segment databases.

- *Broadcast 1:N.* The input relation is sent to *all* other segment databases. The common use case for this operation are join predicates other than equi-joins where one relation is distributed on some column and the other is replicated.

All segment databases execute the same plan and exchange data via the distribution operators as part of the processing. Encapsulating the distribution logic into self-contained operators enables concise reasoning over query plans. The data exchange is pipelined, i.e., data is exchanged whenever available.

Figure 2 (a)–(c) illustrates the different distributions and their effect on query plans for a simple query. Consider the following query over the standard TPC-H Benchmark schema:

```
SELECT lineitem.*
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
```

Figure 2(a), shows a query plan for the case where the table `lineitem` is distributed on `l_oderkey` and `orders` on `o_orderkey`, i.e., both tables are already co-located and the join can be executed immediately on all segment databases in parallel. The results are gathered on the master and returned to the client. In Figure 2(b) a query plan is shown

for the case where the `orders` table is distributed randomly. In order to achieve the correct join result, the table needs to be aligned properly, i.e., all data of `orders` needs to be redistributed on `o_oderkey`. Finally in Figure 2(c), a query plan is shown for the case where both tables are distributed randomly; in this case both tables need to be redistributed to be aligned on the join keys.

The example underlines that the query plan makes up for incorrect or insufficient distribution of data. However, redistributing data as part of processing the query comes at additional cost in terms of running time as well as network bandwidth. We can expect plan (a) to outperform plan (b) significantly and a further significant difference in performance between plan (b) and plan (c).

While this example illustrates the basic concept, the query optimizer actually takes distribution into account as part of *all* of its optimization decisions. In particular, when determining the join order for queries involving more than two tables or derived tables the distribution of data becomes a substantial contributing factor in the optimization and preserving data distributions or establishing distributions that may benefit several joins is one of the optimization goals. In Greenplum Database all optimization decisions are made in a cost-based manner.

3.4 Fault-tolerance & Replication

Since preserving fault-tolerance during expansion is a key requirement for a production system, we briefly survey Greenplum Database's fault-tolerance mechanism. Greenplum Database supports a 1-safety model that tolerates the failure any given component. All hardware components are redundant, including storage and network infrastructure. The data on the segments databases including the master is replicated, i.e., the data is replicated to a designated mirror. Data replication is accomplished using standard physical replication techniques.

The master runs a fault-detection algorithm checking the health of all segments periodically. If a failure is detected, the system is reconfigured to route traffic meant for a failed primary to its mirror, accordingly. Through alerting mechanisms including SNMP as well as online monitoring tools, administrators can troubleshoot the failed primary and, after resolving the root cause of the failure, may start recovery using the system-side provided tools and integrate the restored primary back into the system.

The interaction with the fault-detection system is orthogonal to query processing—none of the components are aware of their replica. During an expansion, holding up the same guarantees for fault-tolerance is an absolute must.

4. EXPANSION

In this section, we present the actual expansion methodology we developed and implemented, in detail. As we will see, the design addresses the requirements as outlined in Section 2.

The fundamental idea underlying our methodology is this: extend the system initially with "empty" segment hosts and then over time redistribute small quanta of the total data set from their original allocation to segment databases across the entire expanded system until all data has been redistributed. The redistribution process is of low impact and happens in the background while regular query workloads are running.

The entire expansion process is orchestrated by a utility called `gpexpand` that uses documented API's of Greenplum Database [4]. Specifically, there are three major phases the tool aides administrators with:

1. *Initialization.* After new segment hosts are physically added to the existing cluster the new systems are initialized and empty segment databases are spun up on the new segments. After this step completed, the new segment hosts are full members of the system and are ready to be used for query processing, data loading, etc.

2. *Redistribution.* The pre-existing data set is redistributed over a—potentially extended—period of time by redistributing individual tables one-by-one according to a pre-defined schedule. `gpexpand` manages this process and allows administrators to prioritize, monitor, pause, and/or resume the distribution process at a rather detailed level to assure the expansion is performed in the background without affecting regular user workloads.

3. *Finalizing.* Once the actual redistribution of data is completed auxiliary tables used for scheduling and prioritization are removed.

As mentioned above, `gpexpand` is primarily provided for convenience. All steps of the expansion process can be executed manually. In the following, we describe the individual aspects of each phase, the API's it uses, and its practical relevance in detail.

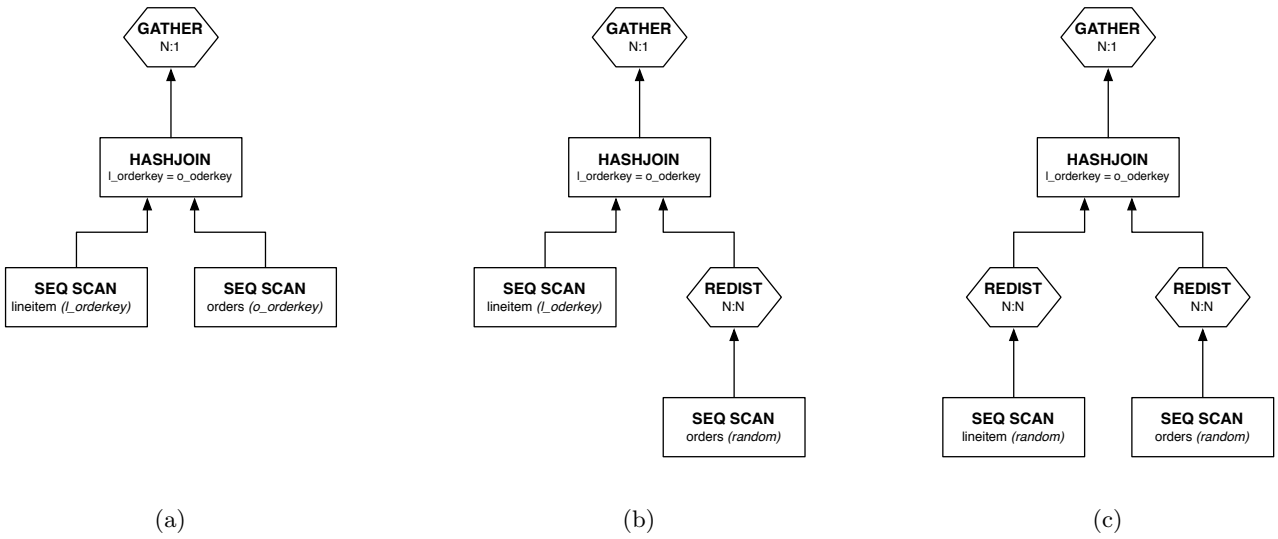
4.1 Provisioning and Initialization

The preparation for an expansion is by far the most time-consuming as well as the most labor-intensive part of the process.

Provisioning. The biggest challenge in this phase is the technical build-out of the extended system: obstacles that need to be overcome include space constraints in terms of rack space, cabling issues, etc. Once fully assembled, the new servers are burnt in using stress test tools provided with the product suite. At this point, the new hardware simply sits physically close to the existing system but is not integrated in any way with the cluster. Stress testing the new hardware ensures defective drives are detected and can be replaced before going into production. In addition to stress testing tools, Greenplum Database also comes with a number of check tools that examine the installation, detect performance anomalies and check for proper configuration of the underlying operating system.

Initialization. With the new servers in place, the next step is to install the database software and initialize the new segment hosts. This step is facilitated by `gpexpand` and administrators are guided through an interactive interview process that determines the hostnames of the additional segment hosts and the layout of primaries and mirrors. Alternatively, the extension can also be configured using special configuration files—this is particularly useful if the expansion concerns several 10's of machines and interactive configuration may be error-prone.

Once configured, all segment databases are initialized using the master database catalog as a template for all new



(a) (b) (c)

Figure 2: Query plan for equi-join with different distributions of input tables.

segment databases. The original distribution policy information is saved off and the metadata for all tables is modified to indicate that the table is distributed randomly. That is, all tables can be queried or modified across *all* segment databases afterwards as if they had a skewed but unknown distribution. During this step, the system stalls all incoming connection or write activity to ensure a consistent copy of the catalog across the new segment databases. In addition, auxiliary tables are created to capture the current status of the expansion listing all tables that have yet to be redistributed.

At this point the system is effectively extended to include the new segment databases and is fully operational. However all data is still located only on the pre-existing system and is nominally distributed randomly. Regular query workloads, loads, etc. can resume at this time. Note, that new tables will be created on *all* segment databases including the new ones.

As mentioned above, Greenplum Database does not require the hardware to be homogenous meaning the new servers may be configured differently, e.g., different number of CPU's, different disk capacity, etc. For simplicity, we assume all servers to be identical to those in the existing system in the following.

4.2 Establishing Distribution Policies

Once the system is back in service, new tables created for data loads immediately leverage the new segment databases.

Data Roll-off. Most data warehouse applications load data using daily partitioning for the fact tables. This means for the expanded system that new data will be distributed optimally within the expanded system using the distribution policy of the original table. As older data is rolled off over the next days and weeks, the data distribution automatically converges to the original design: all new tables are distributed correctly, older tables are successively deleted. In many application scenarios a readjusting due to roll-off of data is sufficient. However, certain tables are not partitioned and not subject to a small retention window, e.g., dimension tables. These tables need to be redistributed to span all segment databases and to re-establish the original distribution policy in order to achieve the desired performance.

Redistribution. In order to redistribute data we needed a simple and robust primitive, ideally of general usefulness, that allows us to redistribute individual tables one at a time to the desired distribution policy.

As the core primitive of the redistribution of data we chose to extend the conventional `ALTER TABLE` syntax to allow administrators to modify the distribution policy of a table and redistribute its data implicitly. The following example restores the distribution policy of `linitem` to be distributed by `l_orderkey` and rebalances its data across all segment databases:

```
ALTER TABLE lineitem
SET DISTRIBUTED BY (l_orderkey)
```

This variant of the conventional `ALTER TABLE` command is fully integrated with the existing DDL framework within Greenplum Database and provides the same transactional consistency as other alterations. Internally, it performs the following tasks:

1. a new temporary table extent is created using the same schema as the original table;
2. all data is read from the original table and redistributed using the standard distribution operators as shown in Section 3 and inserted at the target segment databases as determined by the new distribution policy;
3. all indexes are rebuilt;
4. the metadata of the temporary table is swapped with the one of the original table in the catalog;

The entire sequence of steps is an atomic unit as it is executed within a single transaction. Since it leverages the standard components of the query processor and storage layers all operations provide the same fault-tolerance as regular

operations. During the redistribution, the process holds a table lock to prevent modifications of the table.

Augmenting `ALTER TABLE` in above way has proven to be a very useful tool in general for a number of maintenance operations way beyond expansion only. For completeness, we also cover some special cases here: Changing a table's distribution to be randomly distributed simply wipes out the distribution policy and does not move any data:

```
ALTER TABLE lineitem
SET DISTRIBUTED RANDOMLY
```

In order to redistribute tables that do not have a distribution policy, i.e., are supposed to be randomly distributed, reorganization of the table must be requested explicitly:

```
ALTER TABLE nation SET
WITH (REORGANIZE=TRUE);
```

The latter redistributes all data according to the current distribution policy.

4.3 Query Processing

In order to achieve online or near-online expansion, it is important to resume regular query processing immediately. Depending on the amount of data stored in the database at the time of expansion, waiting for all data to be redistributed is usually not an option.

Modifications to the distribution policy do not affect the query optimizer's ability to create a valid query plan, see Section 3. Although, plans for different distribution policies may differ vastly in performance as we have illustrated earlier. That means no expansion-specific changes need to be made to either optimizer or executor.

Rather, query processing over tables with modified and distribution policies is fully transparent. Its performance characteristics are straight-forward and simple to understand which facilitates troubleshooting.

4.4 Scheduling

During expansion, query performance is degraded because (i) tables are not distributed optimally and (ii) system bandwidth is used to redistribute the data. However, usually not all data is equally "hot" in terms of usage. It is desirable to distribute hot data sets first and prioritize cold data sets lower.

Besides temperature a whole set of other considerations are important when it comes to scheduling the actual expansion of individual tables. To this end `gpexpand` provides options to

- prioritize data sets; hot data sets can be redistributed first
- control degree of parallelism, i.e., number of tables to redistribute simultaneously
- pause/resume redistribution to work around scheduled loads/reports, e.g., redistribute low priority data only during off-hours
- indicate progress

All parameters and options that describe the status of the expansion are stored in auxiliary tables in a dedicated schema inside the database itself. Table 1 shows a simplified version of the schema of the tracking table including descriptions of the columns. During initialization the tool inserts one row per user table into the tracking table. Besides the status of the expansion, we also track parameters that are required to redistribute the table correctly such as the original distribution policy.

Table 1: Schema of expansion status table; used by `gpexpand` to track status per user table.

Name	Description
`oid`	unique identification of user table;
`distribution_policy`	original distribution policy for the table; includes column names and oid's; required to reconstruct original distribution
`rank`	rank determines order in which to expand remaining tables; tables with lowest rank are expanded first; enables prioritization according to business value or frequent access patterns
`status`	status of expansion; possible values are `NOT STARTED`, `IN PROGRESS`, `FINISHED`
`last_updated`	timestamp of last change of status
`expansion_started`	timestamp at start of `ALTER TABLE` command
`expansion_finished`	timestamp at completion of `ALTER TABLE` command
`source_bytes`	disk space occupied by original table; used to estimate progress

After initialization, administrators can modify this table to adjust the priority with which a table is expanded. In case of data roll-off, where old data simply ages out of the system and no redistribution of certain tables is needed or desirable, administrators simply delete the row pertaining to the table in question.

`gpexpand` runs SQL queries against these tables to determine what tables to redistribute next and issues `ALTER TABLE` commands accordingly. The manipulation of the tracking table is transactionally consistent. That is if either the database or `gpexpand` fail the redistribution can be simply restarted. The database is at all times transactionally consistent. The expansion is complete when the status column in all rows in the tracking table has been set to `FINISHED`.

By providing timestamps that indicate when the redistribution of a table was initiated we address one of the most important operational issues: together with the size of the original table administrators can extrapolate and determine expected time of completion.

On the command line of the utility administrators specify the maximum time `gpexpand` may use to redistribute tables. When this timeout expires, no further redistributions are started and the current command is rolled back. This enables administrators to set strict schedules, e.g., to use off-hours for redistribution but stop data movement before, say, nightly loading begins. When restarted at a later point in time, `gpexpand` will automatically pick up where it left off when it was stopped.

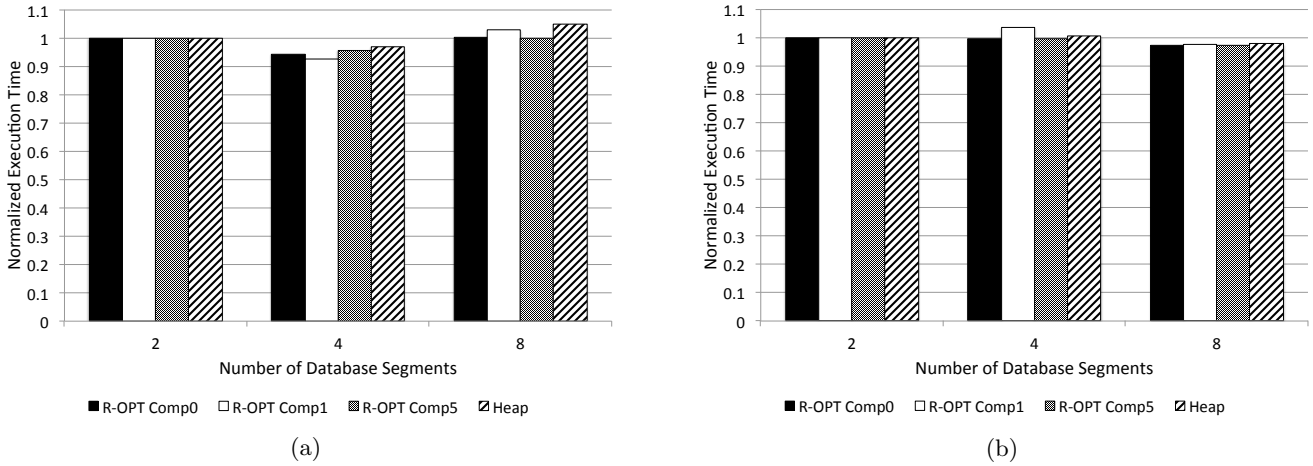


Figure 3: Scalability of redistribution of data for different table types using three different database cluster sizes; (a) TPC-H data, (b) customer data.

The robustness of this approach provides much needed flexibility for a long-running expansion process: redistributing 100's of TB of data may take up to several weeks if the continuous redistribution in the background is run at very low process priority. Being able to interrupt the redistribution process at any given point in time together with the ability to modify priorities and simply continue with the expansion of the remaining tables allows administrators to work around unforeseeable events.

Currently the user surface of `gpexpand` consists of a command line interface and database tables that can be queried directly using SQL. However, these abstractions should make it rather simple to create graphical user interfaces to facilitate configuration and progress reporting in the future.

5. PERFORMANCE EVALUATION

The previous section underlined that expansion of a production data warehouse is a rather convoluted process that depends on a large number of factors besides the size of the original system, the size of the extended system, and the data that needs to be redistributed. This makes it difficult to provide a meaningful performance evaluation.

Nevertheless, in this section we attempt to shed light on some of the most performance critical elements of the expansion process, namely the performance characteristics of redistributing individual tables which focuses on the scalability of our approach. Lastly, we demonstrate the performance impact an ongoing expansion has on a TPC-H query workload.

Hardware Setup. All experiements below were conducted on set of $8 + 1$ Sun Microsystems X4540 Sun Fire "Thor" servers with 2×4-core AMD CPUs, 32 GB main memory, 20 TB disk space, and 4 network interfaces. Each segment host is configured to hold 8 primary and 8 mirror segment databases. The per-server configuration corresponds to one of the popular reference architectures used by Greenplum customers in the past.

Depending on the experiments we configured database clusters of $2 + 1$ (2 segment hosts + 1 master), $4 + 1$ and $8 + 1$ machines with 2×8 segment databases each.

Test Data. To motivate the specific choice of test data it is helpful to review the different storage types Greenplum Database supports first: besides the standard heap-based table type, Greenplum Database also offers read-optimized tables (R-OPT) that support `zlib` compression, c.f. [6]. The most frequently used compression levels at customer sites are 0, 1, and 5. Compressed tables trade off a smaller disk footprint with additional CPU requirements. Therefore, it is important to understand if the different resource demands influence the performance of data movement that would be relevant for redistribution of tables. Heap tables are typically used for dimension tables, R-OPT tables are commonly used for fact tables.

In the experiments below we use both TPC-H data and customer data. We use the `lineitem` table which represents the fact table of the TPC-H schema; we use instances of size 50 GB, 100 GB, and 200 GB, respectively.

Unlike the TPC-H data, the customer data contains more text and is therefore more amenable for compression. The data compresses approximately at a ratio of 7:1 when using Greenplum Database's R-OPT tables with compression level 5 (R-OPT/5). We use samples of sizes comparable to the TPC-H data: 120 GB, 240 GB, and 480 GB, respectively.

5.1 Scalability of Approach

We developed our approach with the aim to be able to expand very large clusters in an online fashion. Therefore, one of the most important criteria to evaluate our method on is scalability.

Initially, we evaluate the scale-up of the basic query processor and storage components involved in the distribution of data. Data passes 5 conceptual components as part of this experiment:

1. read data at source segment database
2. uncompress as necessary
3. send/receive data over the interconnect
4. compress data as necessary
5. insert data at target segment database

Since we are leveraging the regular query processor without taking short-cuts on a lower level of the system, the data needs to be unpacked in the heap and R-OPT/0 cases and, in addition, compressed and uncompressed for R-OPT/1 and R-OPT/5. While using the query processor reduces software complexity markedly, we expect it to be heavier on the CPU requirements. Consequently, we expect this experiment to scale-up near-optimal.

Scale-up of N:N Redistribution. In the first set of experiments, we redistribute data from N source segment databases to N target segment databases.

In each experiment we used 3 different system configurations with 2, 4, and 8 segment hosts. To measure scale-up we need to use proportionate amounts of data, i.e, 50, 100, and 200 GB of TPC-H data as well as 120, 240 and 480 GB or customer data, that is, when doubling the system we also double the amount of data. Recall, that for this experiment the data is originally distributed across N hosts and gets redistributed across N hosts.

In Figure 3 we show results for different system sizes and different storage types. In 3(a), results for TPC-H's `lineitem` data is shown, in 3(b) results with customer data. The results indicate near-optimal scale-up close to 1. In all cases, the results are within less than 10% of the optimum. Also, all table types scale about equally well.

Scale-up of N:2N Redistribution. In the next set of experiments, we vary the number of segment databases and examine the scalability of our approach when distributing data from N to $2N$ segment databases. This corresponds to the data movement when expanding a system by 100%. In Figure 4 we present results for the redistribution of 480 GB of customer data from 2 to 4 segment databases and 4 to 8 respectively. Again, we examine the different table types.

For this experiment we first loaded the data using a 2 node cluster and expanded it to 4 nodes. In the second experiment we loaded the data using a 4 node cluster and expanded it to 8 nodes. The results are shown in Figure 4. The graph shows near-linear scaling also for this experiment; again, for all different storage types.

In addition, the actual data transfer times are in line with the times expected for loading data from external data source such as an MPP loader; this enables relatively accurate predictions concerning the total transfer time given a certain amount of data.

5.2 Query Performance

In our final set of experiments, we examine the impact of an ongoing expansion on a query workload in terms of overhead due to misaligned data distributions.

Starting with a cluster with 4×8 segment databases, we baseline our experiment by running the full suite of TPC-H queries. Then over the course of an expansion and full redistribution of all data to 8×8 segment databases, we re-run the TPC-H query suite after several individual redistribution operations. We distinguish 2 redistribution patterns. First, complete redistribution of all partitions for each table, one table at a time, and, second, simultaneous redistribution of sets of partitions from each table at a time.

1. The order in which we redistribute the tables corresponds to the sizes of the tables beginning with the fact table.

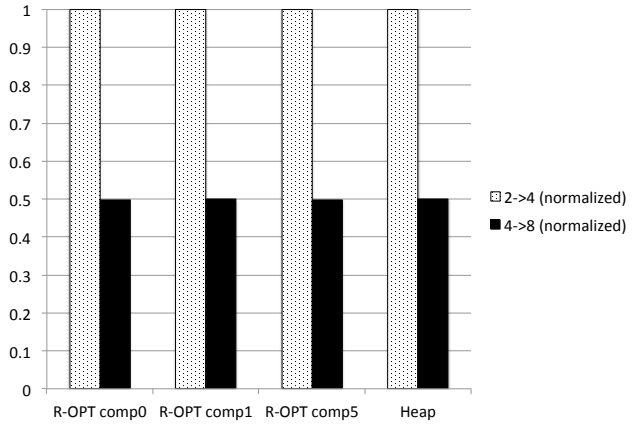


Figure 4: Performance results on expanding a table of 480 GB of customer data from 2 to 4, and from 4 to 8 nodes.

In Figure 5 the complete running time for the TPC-H query suite after each redistribution is shown. As expected, the running time is noticeably longer immediately after the expansion as all tables are marked to be distributed randomly (first data point, about 20% slower). The performance hit is mitigated by the fact that the expanded system has double the number of segment hosts that participate in a query which help with non-trivial queries with numerous joins such as Query Q9.

After redistributing `lineitem`, the running time drops significantly, and is at par with the initial running time. After the complete redistribution of `orders` the running time is already below 60% of the original time. With every further redistribution of a table, the running time approaches 50%, as expected for a system double the size of the original one. For the TPC-H query set, we expect near-linear speed-up when doubling the system in size as almost all individual queries scale perfectly.

2. For the second redistribution pattern, we expand 1, 2, 4, 8, 16 partitions from each table between query runs. This redistribution pattern is more favorable as partitions of different tables that pertain to the same time ranges are redistributed simultaneously. In this case, we see a quicker drop in running time early on and similar convergence toward the expected 50% in elapsed time as all tables are redistributed. See Figure 6.

The last experiment quantifies the impact of unfavorable distributions of tables on a standard query workload. Note that increasing query complexity mitigates the impact of randomly distributed partitions as the query may implement multiple redistribution operations anyways.

In addition, by using query prioritization the actual redistribution can be performed in the background to further reduce the impact on an ongoing query workload once the most significant tables have been redistributed, see [9].

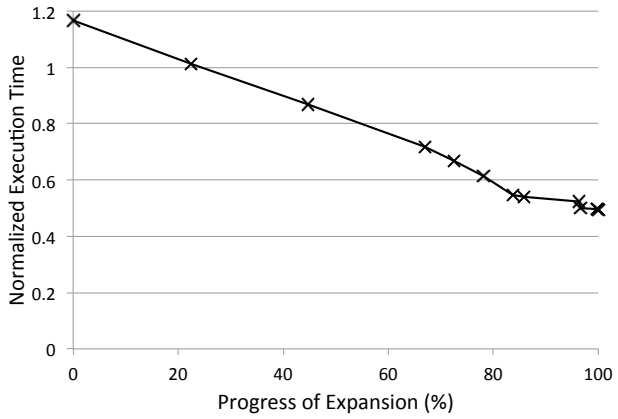


Figure 5: TPC-H query response times during table-by-table redistribution.

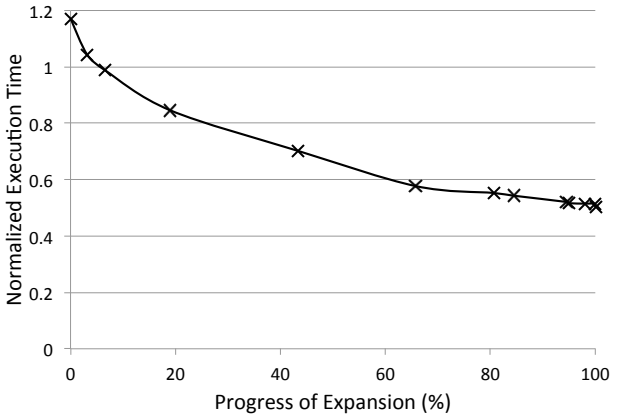


Figure 6: TPC-H Query response times during simultaneous redistribution of all tables.

6. DISCUSSION

Our expansion technique is a careful trade-off accomplishing online expansion on the one hand and using robust and semantically sound primitives on the other hand. More elaborate solutions are conceivable—e.g., distributing data on a finer level of granularity etc. However, we found customers and database administrators clearly preferred the model presented here for its clarity and control: at any point in time, location of data and system behavior are easily discernible and comprehensible for administrators and users alike.

6.1 Evaluation against Desiderata

In Section 2, we enumerated the requirements we originally set out to satisfy. Our approach addresses *scalability* requirements for resulting capacity as well as the *performance* goals for the expanded system naturally in that the final configuration is virtually indistinguishable from a system that was built out on a platform of post-expansion size.

Replication and *disaster recovery* are fully orthogonal to our approach and at no time different than during regular operations.

Our approach scores particularly high on *transparency* and *configurability of the process*: all steps are simple to comprehend for administrators. At any time during the expansion, the system is always in a configuration that could also be created manually by distributing tables in a specific way. As a result, performance expectations can be set accordingly. As we learned from customer interaction, the importance of this property cannot be overstated.

6.2 Disk Space Requirements

Since our approach uses a transactional `ALTER TABLE` command subsequently on individual tables or partitions, it requires extra disk space proportionately to the largest table or partition that is redistributed.

For example, when expanding a system from 20 to 40 segments, the redistribution of a table of size 1 TB requires 50 GB per segment prior to expansion, and 25 GB in the expanded configuration. Given today's multi-TB configurations, the extra disk space needed is negligible.

More generally, the required additional space on all segments is strictly less than the size of the largest unit. In practice, the largest partitions or tables are in the order of less than 5% of the total capacity. Accordingly, this approach will require additional disk space of less than 5% of the total capacity. The extra capacity needed can be computed accurately in advance—the utility provides this information up front—to facilitate planning accordingly.

Since continuous capacity planning is one of the key responsibilities of every administrator, this behavior, together with the ability to forecast space requirements accurately, has been greatly appreciated by operators.

6.3 Lock Contention

`ALTER TABLE` acquires locks on individual partitions during the redistribution process. This affects access to tables in the same way various common and frequently applied DDL operations affect access.

In practice, we have found these lock periods *not* to be noticeable for users for two reasons:

1. Due to the highly partitioned nature of a data warehouse database schema, even the largest table or partition is generally small. Hence, the time during which locks are held for redistribution is short and similar to that of any of a variety of maintenance operations. In most cases the time needed to redistribute a partition is in the order of a few 10's of seconds.

2. Updates are infrequent in a data warehouse environment. In particular, fact tables are almost exclusively loaded in an append-only fashion. Updates to dimension tables are of low frequency, in general.

6.4 Customer Application Scenario

The ultimate test for such an expansion method remains the deployment in production systems at customers' sites.

The expansion mechanism we present in this paper has been developed in close collaboration with several marquee customers. Since its first release in Greenplum Database 3.3, the technique was deployed in a number of customer accounts on production systems—most recently in an expansion from a 12 to a 18 node cluster. The total size of the data set was about 260 TB. The customer successfully leveraged the scheduling facilities of `gpexpand` to avoid reorganizing tables during peak hours, though avoiding peak usage times as precaution rather than actual necessity.

Over the course of several weeks, this customer redistributed all data with no discernible impact on the regular query workloads and no additional disk space or other resource requirements.

7. SUMMARY

Expanding a large production data warehouse is a challenging maintenance operation for any database administrator. In conventional database systems an expansion entails significant downtime, the operation is of high risk, and a logistic feat that is poorly supported by tools.

In this paper we state a set of desiderata that need to be met in order to address the problem satisfactorily in practice. We presented the online expansion mechanism we implemented in Greenplum Database. The foundation for the mechanism is a simple, highly robust and transactionally consistent redistribution technique, fully integrated in the existing DDL framework and exposed as a variant of the `ALTER TABLE` command. We developed a tool based on this primitive to enable sophisticated scheduling and various controls for the database administrator.

Our experiments show this mechanism scales well, is of relatively little impact to ongoing workloads, and enables expanding a data warehouse over an extended period of time, e.g., days or even weeks. Our experience in the field corroborates the in-house experiments, validating our approach.

Acknowledgements

The authors would like to thank all members of Greenplum Engineering who contributed to this work.

8. REFERENCES

[1] Amazon.com. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2`.

[2] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. In *Proc. VLDB*, pages 1481–1492, 2009.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symp. on Operating Systems Design and Impl.*, pages 137–150, 2004.

[4] EMC Data Computing Division. Greenplum Database Admin Guide. `http://gpn.greenplum.com`, 2010.

[5] G. DeCandia *et al.* Dynamo: Amazon's Highly Available Key-value Store. In *Symp. on Operating Systems Principles*, pages 205–220, 2007.

[6] J.-L. Gailly and M. Adler. Zlib. `http://www.zlib.net`, 2010.

[7] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32:422–469, December 2000.

[8] A. Lakshman and P. Malik. Cassandra A structured storage system on a P2P Network. `http://on.fb.me/PwGBa`, 2008.

[9] S. Narayanan and F. Waas. Dynamic Prioritization of Database Queries. In *Proc. ICDE*, pages 1232–1241, 2011.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, pages 1099–1110, 2008.

[11] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41:14:1–14:136, July 2009.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, pages 17–32, 2001.

[13] F. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *Proc. BIRTE*, 2008.