An Evaluation of Storage Load Balancing Alternatives for Database Systems

Nickolay Tchervenski

MMath Essay
David R. Cheriton School of Computer Science
University of Waterloo

February 2008

Contents

Acknowledgements

I would like to thank my supervisors Professor Ashraf Aboulnaga and Professor Frank Tompa for their continued support, help and mentoring. I thank Professor Kenneth Salem for being my essay reader. Last but not least, I thank them for leading some of the most interesting and engaging classes that I have attended in a university.

Dedication

I dedicate this work to my daughter Victoria and wife Denitza for their support and belief in me.

1. Introduction

Modern relational database systems (DBMSes) are often a critical part of the IT infrastructure for many enterprises. In many cases, the success of an IT system is highly dependent on the performance of its database systems. The choice of how database objects, e.g. tables and indexes, are assigned to disks is often a significant factor in determining the I/O performance of the system. It is a common practice for DBAs to rely on fully striping objects across all available disks in order to achieve I/O parallelism. Similar techniques are also suggested in user manuals and are implemented in automatic storage features, for example in DB2 [8]. While this is generally a useful rule of thumb, for queries that involve co-access of two or more large objects, e.g., a merge join of two tables, such practice can be far from optimal due to the large number of random I/O accesses incurred on each disk drive. The problem arises due to the *seek time* of disk drives [10]. This seek time delay is incurred each time the read/write head of the drive has to move to access a particular section of the disk. Naturally, the more random the I/O is, the more time is spent seeking rather than transferring data. One way to decrease object co-access for two objects, is to place these two objects on different disk drives. In DB2, for example, database objects are stored in *tablespaces*, which comprise one or more *containers* on disk. Each container is a file or a raw disk, and all the data in a tablespace is striped across all of the containers in that tablespace. Thus, to solve the co-access issue between two objects located in different tablespaces, all that is necessary is to place the containers of the two tablespaces on different sets of disks.

Balancing the load among disks can also be done at the storage level, typically by re-mapping blocks of pages from one disk to another to decrease the load on the former disk. It is worth noting that such a load-balancer cannot effectively separate two tables with high co-access unless these two tables are infrequently updated. Any new pages created for the two tables could end up on the same physical disk. Only a solution that can recognize the high co-access at the tablespace level can effectively separate the two tablespaces.

In this essay, we propose three methods for performing the basic operation required to balance the I/O load among disks in a tablespace-aware manner, namely moving a container from one location (disk) to another. Two of our proposed methods work from within the DBMS. We are not aware of any current DBMS that provides such functionality. We have implemented our proposed methods on IBM DB2 version 9.5 (Viper II), however the proposed methods should be easily applicable to most other commercial or open-source database systems.

The first method proposed in this essay is to move the file container from one location to another, while the tablespace that this container belongs to is quiesced to a read-only state. While DB2 does not support moving containers while it is online, or even offline, minimal code modifications were made to DB2 to add such a feature. The second method proposed is to utilize existing DB2 functionality to add a container to a tablespace at the new location, rebalance the tablespace, drop the container from the old location, and then rebalance again. Due to the rebalancing being done twice, this is not an optimal solution and is in fact not exactly an operation that moves a container from one location to another. On the other hand, this is the only solution that can be done at the database level by DB2 DBAs at this moment. The third method proposed is to utilize storage virtualization, like the *Logical Volume Manager (LVM)* of Linux systems, to move the logical volume that holds the container from one physical disk to another. This is the easiest solution of all and is most transparent. However, to provide for per-container granularity, each logical volume has to hold only one container. This adds complexity in managing the virtual storage and also couples the virtual storage management to the specifics of the database and its layout. In addition to the three methods mentioned above, we discuss several other alternatives whose implementation is beyond the scope of this work.

The rest of this essay is organized as follows. In Section 2, we present an overview of related work. Section 3 presents the necessary background and our problem definition. Section 4 presents the different solutions compared in this work. In Section 5, we discuss the applicability of our proposed methods to autonomic database systems in general. Section 6 provides details on our experimental evaluation. Section 7 concludes.

2. Related Work

Agrawal et al. [1] show that spreading data across all drives is in many cases far from optimal. For example, in a system with 8 disks, Q3 from TPC-H is shown to execute 44% faster when the *lineitem* table is spread across 5 disks and the *orders* table is spread across the other 3 disks, compared to the case when both the *lineitem* and *orders* tables are spread across all 8 disks. The main reason for the speed up is that the two tables were on completely separate sets of disks, so no co-access of objects on the same disk occurred, thus minimizing extra seek time. The paper presents an offline method to arrange tablespaces/containers across physical disks to maximize performance for a particular workload.

The problem of determining the optimal storage layout has also been addressed at the storage system level. An offline solution to this problem is presented in [2]. It proposes designs and configurations for storage systems based on traces from database, filesystem, e-mail, and other workloads, taking into account contending I/O performance demands when placing the data.

Scheuermann et al. [6] recognizes the significance of how files are striped across disks and that striping across all available disks is not always best, as it could severely limit throughput by increasing “device-busy time consumed in serving a single request.” On the other hand, placing a file on a single disk does not benefit from parallel disk usage. In addition to choosing the granularity at which files should be striped, the paper investigates dynamic load balancing by trying to avoid disk bottlenecks by finding and moving hot data from one set of disks to another.

In [4], Lee et al. propose a solution to solving the problem of assigning files to disks in a parallel I/O system, aiming to minimize both the load imbalance across all disks and the service time at each disk. One issue that this work does not account for is that for databases, no two containers from the same tablespace should be placed on the same physical disk.

Autonomic data placement for load balancing, albeit on shared-nothing database systems, has been explored in several papers, such as [5]. That paper proposes a method to analyze changes in access patterns and decide what data to migrate from an overloaded database partition. Compared to our paper, which is designed for non-partitioned databases, a possible co-access of two tables would cause overloaded disks. In the scenario where the two tables are spread across all available disks, it would not be possible to determine which disks are more used than others. However, there are other methods besides disk activity that can be used to determine what two tables have high co-access.

3. Background and Problem Definition

Disk Drives and RAID Performance

Currently, the most popular and cost-effective data storage used in database systems is the hard disk drive. The key specifications of a disk drive are its size, average seek time, and transfer rate. Seek time for a disk is the amount of time it takes for the head to move to the particular sector to be accessed. For this reason, the actual data read/write rate of a disk is highly affected by the seek time of the disk and the placement of the data across it. To give more light into how seek times affect performance, we will take as an example a popular hard disk drive from Western Digital [10], with 8ms seek time and 98 MB/s transfer rate. A simple calculation shows that by doing a single seek taking 8ms, almost 1 MB of data could have been read during this time. According to the specifications, full-stroke seek takes 21ms, which means that seeking from one end of the disk to the other would slow data transfer even more. Doing seeks every 4 MB of data, which is essentially a thousand 4K pages of data, could slow the transfer rate by as much as 25% to 75%, depending on the seek length. For these reasons, locality of data is a very important factor in database performance and is one of the reasons why data pre-fetching is done by most databases.

A popular method used to improve performance and stability is using *RAID arrays* (Redundant Arrays of Inexpensive Disks). Data is evenly striped across equivalent disks, with possible extra parity disks, to allow for faster transfer rates. This, however, does not improve seek rates because all disks essentially do the same seeks as data is striped across them equally. A simple calculation following up on the example above, with four disks in a RAID array, the seek time will be the same, 8ms, and the transfer rate will be almost 4 times better, which means that a single seek would cause missing 4 times more data. Therefore, with a RAID array of *N* striped drives, reading a single file placed continuously on a disk is *N* times faster than the same operation on a single drive. However, simultaneously reading *N* files spread all over the drive would cause a significant performance penalty compared to reading *N* files, each placed on a single disk.

The slowdown caused by doing seeks is a recognized problem in the industry, and in an attempt to mitigate this, a feature called *Native Command Queuing* has been developed [3] to allow disk drives to process read/write requests in different order to minimize seek time and maximize throughput. While this certainly helps in extreme cases, on average it does not improve the reported average seek time for the disk drives utilizing this technology.

Storage Virtualization and Logical Volume Management

Storage virtualization is used to provide a logical abstraction of physical storage systems. There are several types of storage virtualizations, a discussion of which is beyond the scope of this essay

In this essay, we focus on storage virtualization using LVM, the Logical Volume Manager of Linux. LVM provides an abstraction on top of physical volumes such as hard disk partitions, RAID devices, etc., allowing users to define logical volumes (LVs) on these physical volumes. To the user, LVs can act just as regular disk partitions, which can be used as raw block devices, have file systems mounted, or even used as swap storage.

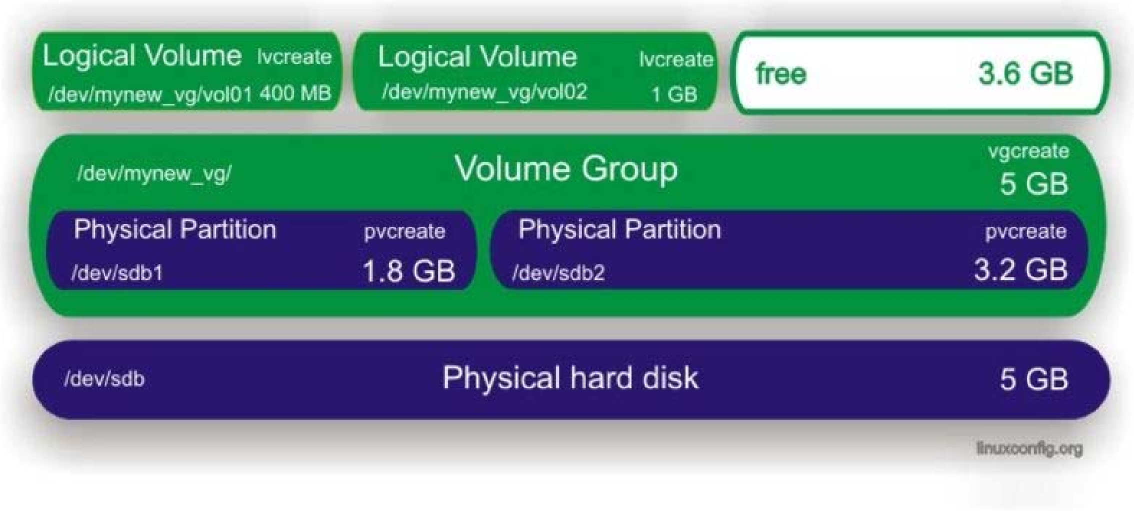

Figure 1 – A sample mapping in LVM from two physical partitions to two logical volumes (image used from [9] under GNU Free Documentation License 1.2)

A sample LVM configuration is shown in Figure 1 above. In this example, a single physical hard disk with two physical partitions is abstracted to two logical volumes of different sizes than the physical partitions. The LVM allows for on-the-fly resizing of the logical volumes and even transparent addition/removal of physical disk drives. In this essay, we are interested in the LVMs capability to resize LVs and move them from one physical partition to another. For example, vol01 of size 400MB above can be moved to the free space on the right, and thus be residing on the other physical partition, /dev/sdb2.

Data Placement in DBMSes

IBM DB2, like most other DBMSes, uses tablespaces to store its objects, such as tables, indexes, materialized views, etc. In DB2, a tablespace is a logical group of containers, where a container is either a file or a disk device. Data in a given tablespace is striped evenly across all of its containers.

Traditionally, to maximize throughput due to parallelism, the number of containers per tablespace is set to be the same as the number of physical disks available. This is done to spread the data as much as possible and thus is a simple way to improve I/O parallelism for the particular tablespace.

Problem Definition

In this essay we discuss methods to transparently move a tablespace container from one location to another, while the database remains online. While we are not aware of such a feature in current DBMSes, and in particular IBM DB2, we believe this is an important step towards an autonomous database storage solution, one aspect of which would be autonomous data placement for performance and reliability. We have implemented and experimented with three methods to move containers from one physical disk to another. The three methods are compared in terms of performance, ease of use, and availability. The first method involved modification of IBM DB2’s engine code, the second involved a less optimal alternative that uses regular ALTER TABLESPACE SQL DDL statements to add a new container at the new location and then drop the container at the old location, and the third method involved using storage virtualization, like LVM, where a logical volume would hold a single container and thus that logical volume could simply be moved transparently from one physical disk to another. Our goal is to compare storage load balancing from within the DBMS to achieving the same result using storage virtualization.

4. Alternative Methods for Moving Storage Containers

In this section, we provide an outline of the three methods we have implemented and experimented with to move a container from one physical disk to another.

Method 1: Adding Container Migration to DB2

Since IBM DB2 does not provide DDL for moving of a container from one location to another, neither offline nor online, we decided to add such a feature to the database engine. Due to the significant code complexity, our aim was to alter the code base as little as possible, while maintaining efficiency as much as possible. After investigating several alternatives, some of which are discussed at the end of this section, we settled on the following way to achieve our goal of moving a container from one location to another:

- make the particular tablespace read-only through a tablespace quiesce command
- copy the container file to the new location. Note that while the copy is being performed, DB2 still uses the old container for read-only queries.
- update internal files and in-memory data structures of DB2 with the new location of the container
- un-quiesce the tablespace
- delete the container from the old location

The obvious limitation of this method is that we make the particular tablespace read-only while we are moving the file to the new location. Later in this section we discuss alternatives that do not suffer from this limitation, but that involve more extensive engine modifications to achieve the same goal. Since tablespace containers are either files or disk partitions, the data can be easily copied to the new location. Once the copy is over, the files and internal data structures of DB2 are updated with the new location of the container, and the container can be re-opened for read/write access. Only after the container at the new location is re-opened by DB2 can the old container be safely deleted.

Method 2: Using the Available DDL Statements

In this method, to effectively move a container from one location to another, we simply create a new container at the new location and add it to the tablespace. Subsequently, we remove the old container. All of this is done transparently to the user and can be done under a read/write workload. For example the following two commands can be used [8]:

ALTER TABLESPACE ts_name ADD (FILE '/newpath/cont5' 1GB)
ALTER TABLESPACE ts_name DROP (FILE '/oldpath/cont5')

Data stored in a tablespace is striped evenly across all of the containers in that tablespace. Thus, adding or dropping a container is an operation that requires rebalancing of the data across the new set of containers [8]. The rebalancing operation is started automatically by DB2 and is run transparently in the background, without affecting read/write queries. However, while the tablespace is being rebalanced it cannot have any of its containers dropped or added.

The rebalancing of data due to the *ADD container* operation above causes data to be spread evenly across all containers. This includes moving pages of data from all the containers to the newly added one. Only after this rebalancing is done can the *DROP container* operation be performed. This will cause another rebalancing to occur, this time to move the data from the removed container to all other containers. The end result is that we “move” the container from the old location to the new one, but data gets shuffled across all containers of the tablespace.

In this method, two tablespace rebalances occur that involve data movement across all of its containers. This is far from optimal as data pages are unnecessarily moved back and forth across containers. Each rebalance takes relatively long time as it is automatically throttled to minimize performance degradation on the system [8]. On the other hand, short of backing up and restoring a tablespace, there is currently no other way to re-configure a tablespace in DB2 with the effect of having one container moved from one location to another. Furthermore, this is the only method that can achieve this while the database is online.

An alternative to this method, as discussed later in this section, is to create a new ALTER TABLESPACE command that can move a container from one location to another. By modifying rebalancing-related functionality, such an operation would involve only a single rebalance. The rebalance operation would only have to move the pages from the old location to the new one. However, such an implementation is beyond the scope of our work.

Method 3: Using Storage Virtualization (Logical Volume Manager)

In contrast to the other two methods that are performed at the database system level, this method utilizes functionality provided by a storage virtualization tool. For this work, we use the Logical Volume Manager of Linux (LVM). A container that is placed in a single logical volume (LV) can have its LV moved from one physical volume to another, using the pvmove command [9]. This is done transparently to the database system. There are several potential problems with this approach:

- **LV management:** to achieve a granularity of a being able to move a single container at a time, there needs to be exactly one container per logical volume. This adds extra complexity in managing so many logical volumes. Furthermore, this couples database management with storage management.
- **Explicit mapping of logical volumes (LVs) to physical volumes (PVs):** it is not currently possible to explicitly control how LVs are mapped to PVs. Furthermore, there are only two ways to specify this mapping: linear or using striping. Striping of LVs across PVs is not useful in our case as it defeats the purpose of separating containers on different physical disks.
- **Handling and tracking automatic volume management:** when a disk is removed from the volume group, all of its physical volumes (PVs) are also removed. Therefore all LVs associated with those PVs are automatically re-mapped to different physical volumes. Such automatic re-mapping needs to be closely monitored and controlled. This further adds to the complexity of properly managing the LVM system.

Other Alternatives Not Implemented

As mentioned above, there are several alternatives that we considered implementing in addition to the three methods that we have implemented. Those that we list below are all at the database level, and are thus similar to Method 1 above.

Rebalancing based on-line algorithm: This method is a combination of the first two methods we have implemented above. Reusing the rebalancing framework in DB2, all the pages of a container can be moved one by one to the container at the new location. This can be done in the background, just like regular rebalancing is done. The implementation would require a new ALTER TABLESPACE operation, which can be called MOVE. For example, the command can be as follows:

ALTER TABLESPACE ts_name MOVE (FILE '/old_location/ts5' '/new_location/ts5')

While re-using a lot of pre-existing engine code, such functionality requires adding the necessary recovery (undo/redo/rollforward) functionality, which made it impractical to implement within the scope of this work.

In terms of performance, this method would be similar to a regular rebalance operation – it would run in the background while the database is still online and can be throttled to minimize performance impact [8]. Furthermore, the level of throttling can be controlled by the DBA.

Update at both locations during the copying of the container: As above, at the database level, a page-by-page copy operation can be performed. To ensure consistency between the two copies, during such an operation any page updates will be performed on both copies of the container. Updates to pages that have not yet been copied over to the new location can be performed at the original location only. When properly implemented, this method has the potential of being simpler than the rebalance based algorithm above and faster on workloads that are not update-intensive. The reason for it being potentially faster is that, in the above solution, when a page is moved from one container to another during rebalancing, its association with a container has to be changed, and this operation consumes additional CPU cycles. This method can also be throttled like rebalance and other DB2 utilities. However, doing updates at two places until the move operation completes is a definite performance hit. Another possibility for this approach is to do it at the file virtualization level, outside the database system.

Utilize Online Backup/Restore: It is possible to utilize techniques used for online backup to copy the pages of the container over to the new location, and then rollforward that container to the present time. This technique would be the most difficult to implement, and it may require making the tablespace read-only during the rollforward phase. Rolling forward through all the log records since the start of the move operation and applying the changes only to this container can prove to be more costly and complex to implement than the other methods described above.

5. Possible Usage in an Autonomic DBMS

For a comprehensive storage load balancing solution, we need the following components in addition to the on-line container move:

- A way of measuring storage load by monitoring the file and/or database systems. This component should be able to gather usage statistics at the file and/or database systems level such that tablespace co-access can be recognized and have its performance impact on the system be estimated.
- A way of deciding based on system/storage load and a set of pre-defined policies when to move a container. This control component, given the data gathered by the storage load monitoring component, needs to be able to determine which two tablespaces would be most beneficial to separate, i.e. have their containers moved to separate physical disks. To be able to make decisions, the component needs to have detailed knowledge of both file system/physical disks specifications as well as database specifics like tablespace to container to file system mapping. Based on this information, system policies and storage monitoring feedback data, the component can decide when and how to move a container to improve database and file system performance.

- A way to log system tuning activities and storage load and be able to produce reports for system administrators. Such information will give system administrators the ability to fine-tune system policies, produce historical data reports, provide service level guarantees and more.

The ability to move containers online has other uses than in complete load balancing solutions like the one described above. Current database systems, such as IBM DB2, have added automatic storage features that relieve DBAs of the need to manage containers for tablespaces [7, 8]. To use this functionality, a database is created with the automatic storage option, and a set of filesystem paths is defined to be used by the automatic storage manager. The database system would then automatically create and manage the containers for the tablespaces using the paths defined for automatic storage. At the present time, data is spread evenly across all the paths, which is not always optimal and can be improved. There is an opportunity for database systems to be able to arrange containers automatically and thus autonomously improve I/O performance.

In general, the ability to move tablespace containers coupled with the ability to move tables and indexes across tablespaces is an important step towards implementing autonomous database systems that automate database layout while the database is online. Based on this functionality, online search algorithms can be designed to find optimal database layout on a running database.

6. Experiments

Experimental Setup

Test Environment: We have implemented the three scenarios described in Section 4 and evaluated their performance on IBM DB2 V9.5. The experiments were conducted in a VMware environment running a 64-bit single CPU CentOS 5 operating system with 1.5GB of memory. The host machine was a 3GHz 2GB dual-core Pentium D905 machine running 32-bit PCLinuxOS. VMware was chosen due to limited resources and mostly due to its flexibility in controlling the test environment, providing us with, for example, easy backup and restore of the initial configuration, configuring system memory availability, configuring hard disk drive availability, and more. To avoid caching at the VMware level, the virtual machine was given direct physical access with write caching disabled to the drives containing the database data.

Database: We created a custom database, SAMPLE, with database memory set to *COMPUTED* and all other memory consumers (bufferpools, sort memory, locklist, etc.) set to *AUTOMATIC*. For all scenarios, the values for bufferpools and sort memory remained relatively stable, and are given in Table 1.

	Size in pages (4k size)
Bufferpool	80-84K (AUTOMATIC)
Sortheap	0.4-0.6K (AUTOMATIC)
Sheapthres_shr	2-3K (AUTOMATIC)
Database_Memory	150K (COMPUTED)

Table 1 – Memory consumption of different memory pools

The database size was chosen to be large enough so that it cannot fit in memory, which would make disk performance irrelevant. The SAMPLE database contained two tables, A and B, with one million rows each. The two tables both had three columns and the corresponding columns had the same data types. The definitions of tables A and B are given in Tables 2 and 3, respectively.

Column	*Data Type*	*Restrictions*
a_key	INT	NOT NULL PRIMARY KEY
a_value	VARCHAR(2000)	
a_price	DOUBLE	

Table 2. Definition of table A

Column	*Data Type*	*Restrictions*
b_key	INT	NOT NULL PRIMARY KEY
b_value	VARCHAR(2000)	
b_price	DOUBLE	

Table 3 – Definition of table B

The two tables were populated with data from the same domains, but the rows of A and B were completely disjoint. The ranges of values for each column of table A are described in Table 4. Table B was generated in a similar manner.

Column	*Range*	*Description*
a_key	1 to 1,000,000	Each value was used exactly once.
a_value	`AA…AAAAAAAAAAAAAAAAAAAA to AA…BBBBABAAAABAABAAAAAA`	Represents the binary representation of all values from 0 to 1,000,000. Where A represents 0 and B represents 1. Each string is 2,000 characters long. Each value is used exactly once.
a_price	0 to 9	a random value

Table 4 – Ranges of values for table A

Tables A and B were populated with the same values for their columns, inserted in the order of increasing value of a_value, but had their primary key values (a_key and b_key) shuffled randomly. The algorithm to populate the tables is presented in Figure 2. This algorithm resulted in each table having a size on disk of 2GB.

```
const int N = 1000*1000;
int a_key[N] = {1,2,3,4,5,…….,1000000};
int b_key[N] = {1,2,3,4,5,…….,1000000};
char a_value[2001] = “AAAA…..A”;

// shuffle the values of  a_key and b_key arrays
//
// the algorithm to shuffle an array of size N is:
//    - randomly pick a number X from 0 to N-1
//    - while( we have picked X already )
//    -          X= (X+1)%N
//    - repeat until all N values have been used

shuffle_values_in_array(a_key);
shuffle_values_in_array(b_key);

for(i=0;i<N;i++)
{
     int a_price = random(10);      // random value from 0 to 9
     insert_record_into_table(“A”, a_key[i], a_value, a_price);

     int b_price = random(10);
     char *b_value = a_value;      // use same value for b, but different key
     insert_record_into_table(“B”, b_key[i], b_value, b_price);

     // increase a_value by 1, as if a_value were a binary string,
    // where A is 0, and B is 1.
     increase_string_alphabetically(a_value, ‘A’,’B’);
}
```

Figure 2 – Algorithm to populate tables A and B

Workload: There were two types of statements run against the database: a read only merge-join query on the two tables and an update statement on each table. The statements are given in Figures 3-4.

```
SELECT a_key, MAX(a_price, b_price)
FROM  A,B
WHERE  a_key = b_key
      AND a_key >  @bottom      -- chooses rows with a_key/b_key between @bottom and @top
      AND a_key <  @top
      AND a_value < b_value     -- decreases selectivity by a factor of two
      AND a_price = @a_price    -- decreases selectivity by a factor of ten
      AND b_price = @b_price    -- decreases selectivity by a factor of ten
 ORDER BY a_key
```

Figure 3 – Merge-join query on tables A and B. Parameters used are: @bottom,@top, @a_price, @b_price

```
UPDATE @table
SET price = @new_price
WHERE a_key = @a_key
```

Figure 4 – Update query for tables A and B. Parameters used are: @table, @new_price, @a_key.

The merge-join query was designed to require reading (@top - @bottom) number of rows from both A and B tables in order of a_key. As described above, the tables A and B were populated in such a way that these rows would not be consecutive. To decrease the selectivity of the query, a_value, b_value, a_price and b_price were added in the predicate clause. For example, a_value is smaller than b_value in about 50% of the cases; a_price and b_price are columns that have only 10 distinct values, which are equally distributed, and thus each decreases the selectivity of the query by a factor of 10. Therefore, if (@top - @bottom) is 10,000, there would be that many rows read from table A and the same number of rows read from table B, however the query would return only 10,000 / (2 * 10 * 10) = 50 rows of data.

In our tests, we ran four infinite closed streams of queries. Each stream had its own database connection and was running the same type of parametric query but with different parameters each time. There were no delays between consecutive queries. The four concurrent query streams we ran were:

- Two concurrent streams of read queries. To increase disk contention, we chose to have more than one stream for running read queries. The parameters @bottom and @top were chosen randomly between 0 and 1,000,000 (the number of rows in A and B) such that @top - @bottom was always 10K. The other two parameters, a_price and b_price were assigned random values from 0 to 9 inclusively. We chose a constant @top - @bottom value to make each two queries run in approximately the same amount of time. We chose 10K as the difference between @top and @bottom for two reasons. First, we wanted it to cover a significant amount of the data from each table, and second, we wanted the query to take a reasonable amount of time, about 3 minutes in our case. If we have a very long query, we would not be able to accurately estimate the performance impact of shorter-term events, like moving a container.
- One stream of update queries on table A and one stream of update queries on table B. These two separate streams represent single-row updates to table A and table B respectively. The @a_key and @b_key parameters were randomly chosen from 0 to 1,000,000 (the number of rows in A and B). The @new_price parameter was randomly chosen from integer values from 0 to 9 inclusively.

Each statement is considered to be one transaction, and we measure performance in terms of *Transactions per Minute* (TPM).

Tablespaces and Disk Layout: Tables A and B were created in two different Database Managed Space (DMS) tablespaces, TSA and TSB. DMS tablespaces are managed by the database system, while System Managed Space (SMS) tablespaces are managed by the operating system. In short, for DMS tablespaces DB2 allocates all tablespace containers up-front and allows for altering the tablespace to add or drop containers. For SMS tablespaces, however, DB2 allocates containers on demand and does not allow for adding/dropping of containers. We chose to use DMS tablespaces for the better flexibility and control of managing containers within a tablespace. Furthermore, it is claimed that DMS tablespaces have better performance in most cases (see "Comparison of SMS and DMS table spaces" in [8]).

There were two identical hard drives on the test machine, which we call HDA and HDB, used exclusively to hold the containers for the two tablespaces. Both devices were Seagate Barracuda 7200.9 (7200rpm) 250GB drives with 8MB cache and an average seek time of 11ms.

We tested three configurations of tablespace to disk mappings. In Configuration 1, TSA and TSB each consist of a single container, CA1 and CB1, respectively, placed on HDA. In Configuration 2, CA1 is placed on HDA and CB1 is placed on HDB. In Configuration 3, TSA consists of two containers, CA1 and CA2, placed on HDA and HDB, respectively, and TSB also consists of two containers, CB1 and CB2, also placed on HDA and HDB. Configuration 2 and Configuration 3 represent disk separation and full striping, respectively. The three configurations are depicted in Figures 5-7. The containers were 3GB each. In the LVM-related scenario, the logical volume size was set to 3.5GB, to allow some extra space for the file system. Physical volumes on each disk were set to be 40GB each.

For our testing, it is important to show how one container is moved from one location (file system) to another, and how it affects the read/update workload during this operation. Thus, the most appropriate scenario is to move from Configuration 1 to Configuration 2, where we just move CB1 to HDB. In our experiments, we use Configuration 3 to show that for this database and workload, full striping (Configuration 3) is inferior in performance to data separation (Configuration 2). Obviously, Configuration 1 is the slowest, as it utilizes only a single disk.

Test Scenario: Our test scenario proceeds in three phases. In each phase, we measure the average performance of the read and update streams (in TPM). The first phase of our test scenario was warming up the database to achieve stable steady state performance while in Configuration 1. We considered that the database had achieved stable steady state performance when there was little variance in performance for the given workload and there were no significant changes in self-tuning memory parameters like sortheap and bufferpool size. All the container moving methods that we tested started from this warmed up state, and thus were expected to have the same initial performance. In the second phase of our test scenario, we transition from Configuration 1 to Configuration 2 by moving the container CB1 from HDA to HDB. We run three different versions of the test scenario, using a different method from those presented in Section 4 to move the container in each version. We measure the time to complete this phase and the read/write performance during the move. It is expected that these performance measures will be different for different versions of the scenario (i.e., different methods for moving the container). In the third phase of our test scenario, we wait until performance reaches a stable steady state in Configuration 2. In the second version of our scenario, which is the one that involved adding and dropping containers and doing tablespace rebalances, we ran through an additional phase that involved transitioning from Configuration 2 to Configuration 3. This was done solely to see what the database performance would be when both tablespaces were striped across all drives.

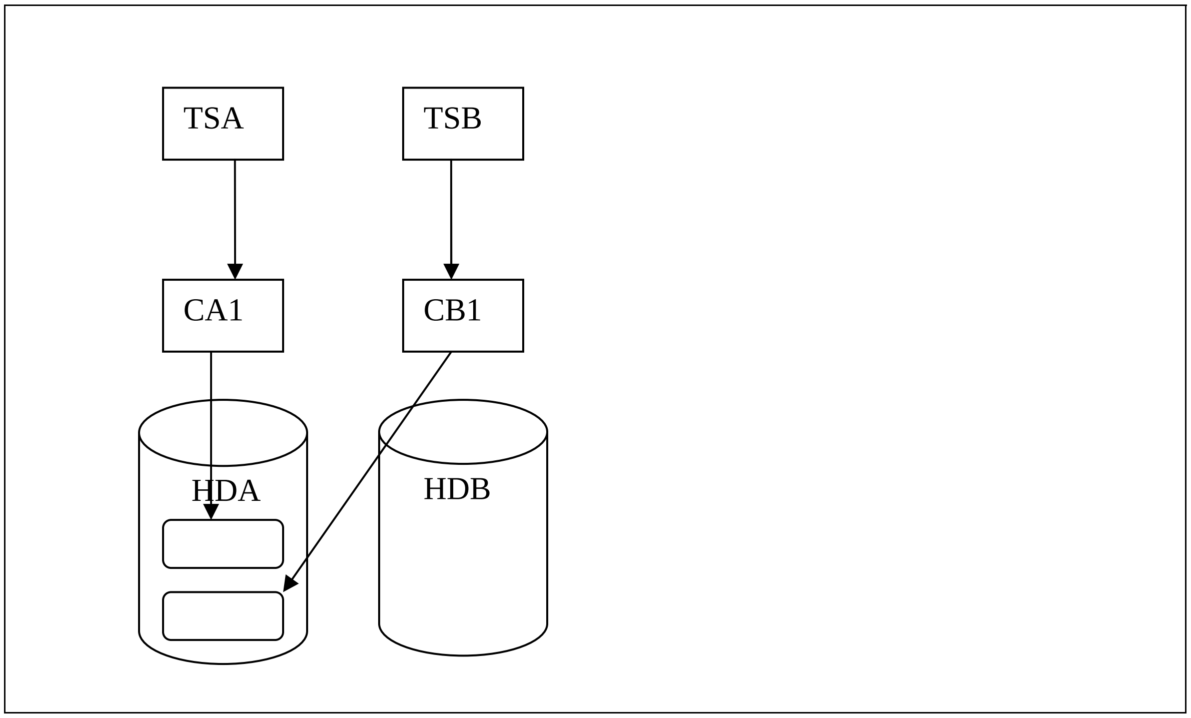

Figure 5 – Configuration 1: Tables A and B are on the same disk drive

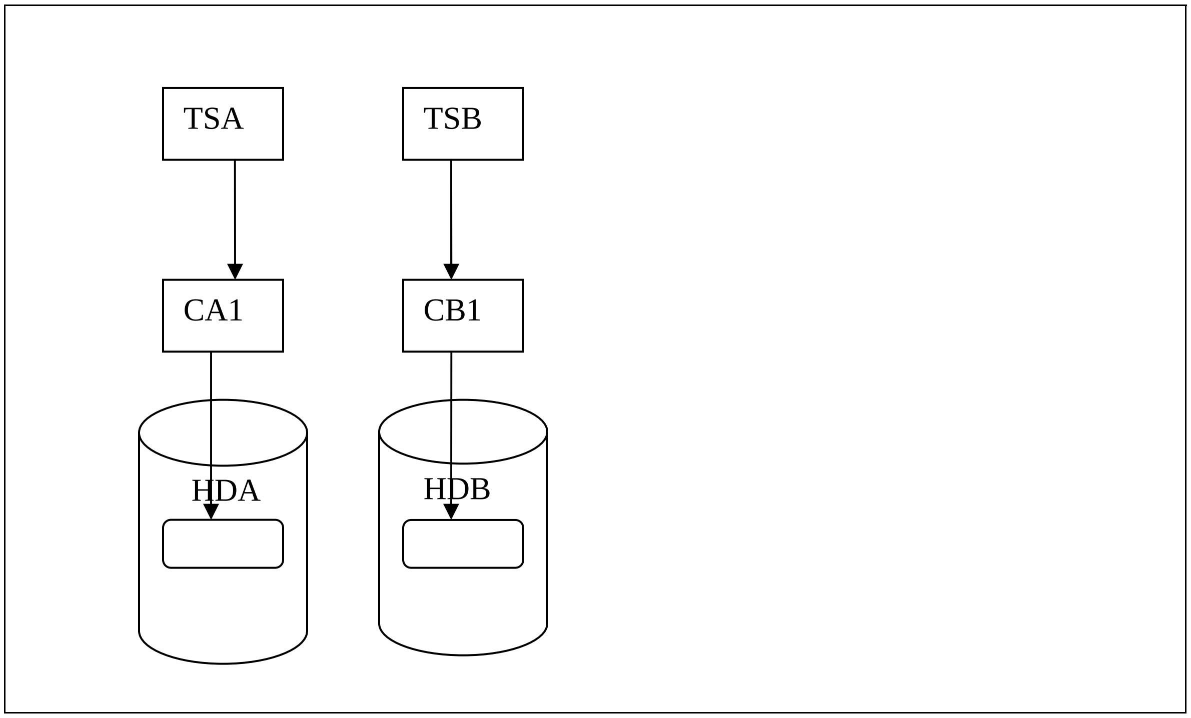

Figure 6 – Configuration 2: Tables A and B are on separate disk drives

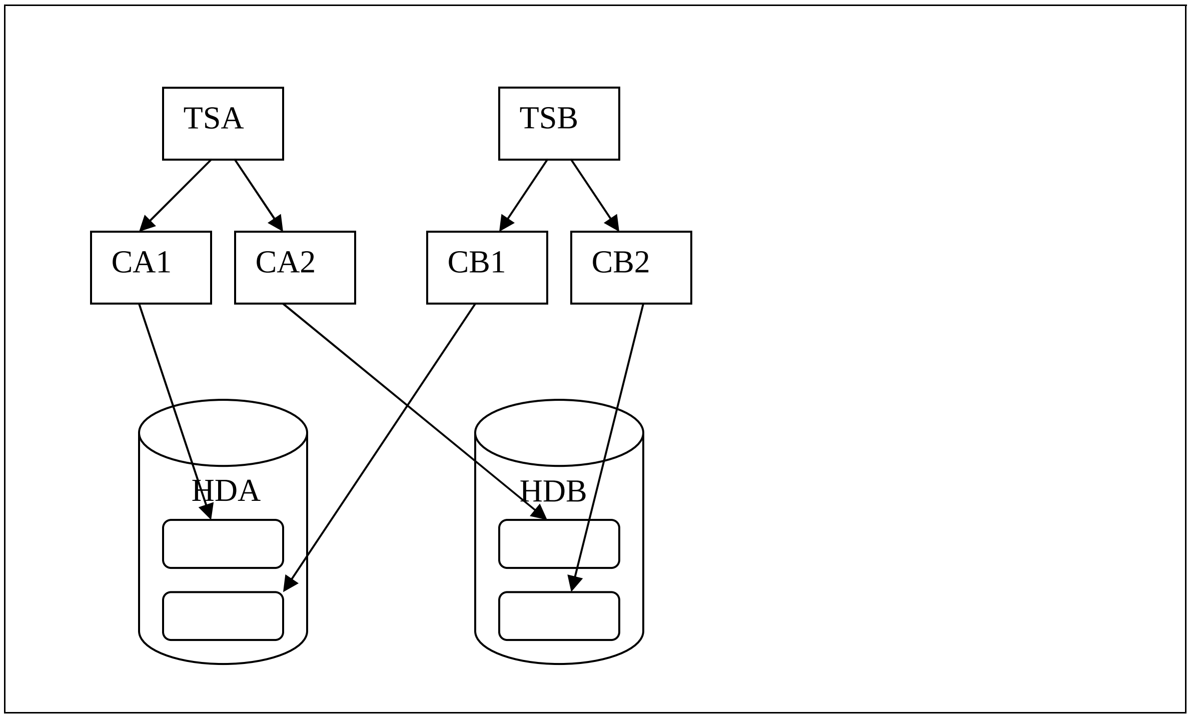

Figure 7 – Configuration 3: Tables A and B are striped across both drives

Experimental Results

[scenario] Query	Avg. TPM after warm-up period	Length of transition period [minutes]	Avg. TPM during transition period	Time to reach stable state [minutes]	Avg. TPM at final state
Scenario 1 – quiesce and copy file over		3		7	
[1] SELECT on A,B	0.3	**	0.25	**	0.8
[1] Update Table A	510	**	550	**	560
[1] Update Table B	520	**	0	**	570
Scenario2 – add, rebalance, drop, rebalance		27 (3 – add, 11 – rebal., 0 – drop , 13 – rebal.)		8	
[2] SELECT on A,B	0.3	**	0.2 first 6 minutes, 0.4 later on	**	0.8
[2] Update Table A	510	**	460	**	540
[2] Update Table B	510	**	430	**	560
Scenario 3 – LVM, move logical volume		23		2	
[3] SELECT on A,B	0.3	**	0.25	**	0.8
[3] Update Table A	515	**	515	**	520
[3] Update Table B	520	**	465	**	690

Table 5 – Experimental results (** means same for all queries in the scenario)

Table 5 presents our experimental results. In our experiment, the update streams produced slightly unstable numbers across scenarios, probably due to the fact that each update was run via a single row update executed through the Linux shell. Since there were about 500 calls per minute in the update streams (500 TPM), the variations in performance across the different scenarios could be attributed to the way that Linux handles shell calls. The delay in the Linux shell may also be the reason why updates do not become twice as fast once we are in Configuration 2. This is in contrast with the SELECT queries that take more than a minute per query to complete, which causes the Linux shell overhead to be negligible.

There are a few points worth noting about the different scenarios. In Scenario 1, during the transition period from Configuration 1 to Configuration 2, the tablespace for table B is quiesced in read-only mode and thus no updates succeed on that table. Also, the time to reach stable state is relatively long due to the fact that once the container was moved it had to be re-opened and thus bufferpools needed to be re-populated. The file system write-cache could have played some role here as well, since the file copy complete might have been reported earlier than the completion of the writes to disk. The effect should have been small however, since the majority of system memory was used by DB2. The average TPM of the read query during the transition period was affected in both the positive and negative directions. It was negatively affected due to the non-throttled file copy operation between the two physical disks. At the same time, it was positively affected by the lack of updates on table A due to the read-only state. The net effect, however, as expected, was an overall slowdown of the select query during the transition period. On the positive side, for this scenario, the length of the transition period was much shorter than that in Scenarios 2 and 3.

Scenario 2 was interesting in that the transition phase consisted of several sub-phases. First, the container was added at the HDB disk. This involved allocating 3.5GB space in HDB for the container. Even though the file was created on HDB and the workload was still being run exclusively at the HDA disk, unexpectedly there was a significant performance degradation during this first part. This performance degradation may have been due to a combination of internal DB2 locking and CPU usage during container creation. On the one hand, since this only took 3 minutes, the performance impact was not severe: the two SELECT queries finished in almost 6 minutes (0.15 TPM) instead of taking 3 minutes each to complete (0.3 TPM). On the other hand, the select queries took 3 more minutes each to execute, as if they were stalled during the container creation. This was not the case however, because the UPDATE queries were only slightly delayed (from 510TPM to 460TPM) during the 3 minutes it took to create the container. In the second sub-phase, rebalancing took place, moving half the pages from the container in HDA to the new container in HDB. The performance of the SELECT queries was relatively unaffected in this phase and was close to normal. Third, the container at HDA was removed, and this operation took DB2 only 1 second to register as complete. However, the actual operation had not completed until after the subsequent 11-minute rebalance ended. That rebalance took the remaining half of the data from the container and moved it to the new container at HDB. During this time, SELECT query performance was relatively stable, similar to steady state levels. It would be interesting as future work to see how this scenario would perform with more disks and containers, where data from all containers needs to be rebalanced to the newly created container. It was surprising to see that the time it took to reach stable state was also long here, 8 minutes. Bufferpools should have remained hot, even during the rebalancing. A possible explanation could be that right after the rebalancing operation we noticed that the *sortheap* memory pool was tuned down, from 200 4K-pages to about 100 4K-pages. It is possible that this caused the slowdown effect.

The extra phase of Scenario 2 was to add one more container per tablespace so that both tablespaces would be striped across both disks. The first container took 3 minutes to add, and after the first add operation finished, the second container was added, which took 7 minutes. It is possible that the second container took more time to add due to the ongoing rebalance operation from the first container and the depleted write cache of the operating system. The subsequent rebalancing completed in 29 minutes, and the performance of the SELECT query streams was just under 0.6 TPM. This is a two-fold improvement over using a single disk, which is expected. However, as predicted, performance is lower than Configuration 2, where the two tablespaces were on separate disk drives so there was no co-access between the tables. The performance of the UPDATE streams was not significantly different between Configuration 2 and Configuration 3.

Scenario 3 was fairly easy to test and proved to be relatively predictable. The transition period was rather long, comparable in length to the one in Scenario 2, where two rebalancing operations occurred. Moreover, it was almost 8 times longer than in Scenario 1, where a non-throttled file copy was used to move the container. The long time it took LVM to move the logical volume from one disk to another may be due to the constant reads and updates being done by the database workload. The logical volume itself was 3.5GB compared to 3GB container size in Scenarios 1 and 2. Furthermore, the actual data in the container was estimated to be about 1.5GB. The workload experienced about 15% slowdown during the logical volume move. The amount of time necessary for the workload to get to stable steady state was negligible in this case, as the database was not affected and thus its bufferpools were intact. Other than some possible caching at the OS level, there should not have been any effects on the database. We recorded the time to reach steady state to be around 2 minutes, which is less than the time a single SELECT query takes to complete.

From these experiments, we can see that LVM performs best. It is quite likely, however, that one of the alternative implementations of Method 1 for moving containers given in Section 4 would be able to achieve better results than LVM. Such an implementation would be based on a single rebalance and throttled container creation, so it should take at most half the time taken for the transition in Scenario 2, and would have had better performance than Scenarios 2 and 3. Care needs to be taken, however, to minimize the time it takes to reach stable state once the data movement is complete. Although our experiments showed some delay in the rebalancing case, we believe it can be avoided in a scenario with a single rebalance.

7. Conclusion

Storage load balancing for database systems can be done at the virtual storage level. In this essay, we investigate doing it from within the database system without requiring a storage virtualization layer. We proposed, implemented, and tested three methods to move a tablespace container from one location to another (one disk to another). We showed that, while storage virtualization may produce good results at the expense of more complex management, it is possible to leave database storage load balancing decoupled from storage virtualization by implementing new forms of rebalancing within the database system that can effectively move a container from one location to another.

Possibilities for future work include testing the proposed methods on other database systems, especially open-source ones. We expect that most of the suggested methods will be feasible on a large range of database systems. Another possible direction for future work is comparing the performance of our methods to a generic load-balancing storage system and evaluating the differences between the two approaches.

8. References

[1] Sanjay Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek Narasayya. Automating layout of relational databases. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2003.

[2] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. In *ACM Transactions on Computer Systems (TOCS),* 2005.

[3] Intel Corporation and Seagate Technology. Serial ATA Native Command Queuing. White paper available at http://www.seagate.com/content/docs/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf, July 2003.

[4] Lin-Wen Lee, Peter Scheuermann, and Radek Vingralek. File assignment in parallel I/O systems with minimal variance of service time. In *IEEE Transactions on Computers,* 2000.

[5] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD),* 2000.

[6] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 1998.

[7] Autonomic and Other Enhancements in DB2 9. http://www.ibm.com/developerworks/db2/library/techarticle/dm-0606ahuja2/#storage.

[8] DB2 Information Center, http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/.

[9] Linux (LVM) – Logical Volume Manager. http://www.linuxconfig.org/Linux_lvm_-_Logical_Volume_Manager.

[10] Performance Specifications of Western Digital Caviar SE Hard Disk Drive. http://www.westerndigital.com/en/products/products.asp?driveid=198&language=en#jump11.