Storage Layout and I/O Performance Tuning for IBM® Red Brick® Data Warehouse

Matthias Nicola
IBM Silicon Valley Lab
mnicola@us.ibm.com

Abstract

Although Red Brick performance depends primarily on the logical database design, the placement and allocation of data files in the disk subsystem is another significant performance factor. There are many choices in the configuration and design of the physical database layout. This article reviews best practices propagated by storage and database vendors and presents measurements that compare storage layout alternatives for Red Brick. This leads to specific I/O configuration and tuning guidelines for IBM Red Brick Data Warehouse.

This article assumes that the reader has basic knowledge of Red Brick Data Warehouse.

Notices and trademarks

The furnishing of this document does not imply giving license to any IBM patents.

References in this document to IBM products, Programs, or Services do not imply that IBM intends to make these available in all countries in which IBM operates.

The information contained in this publication does not include any product warranties, and any statements provided in this document should not be interpreted as such.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
Enterprise Storage Server
DB2
DB2 Universal Database
IBM
Informix
Red Brick

Other products or service names may be trademarks or registered trademarks of their respective owners.

Table of Contents

1 Introduction

Designing performance for an IBM® Red Brick data warehouse does not begin with storage layout and I/O tuning. A great storage layout and finely tuned I/O performance cannot make up for poor join performance caused by inadequate schema or index design. After logical design, you must also consider capacity planning to estimate the number of required CPUs and disks. Both logical design and capacity planning are not addressed in this article.

However, eventually there comes a time to optimize the physical database design and its storage configuration. Because Red Brick is typically used to process complex analytical queries over huge amounts of data, efficient I/O is often an important performance factor. You must map all tables and indexes to a set of database files (called *physical storage units*, or PSUs), which in turn are allocated in the disk subsystem. Ideally, data placement should allow you to meet the following goals:

- To maximize I/O throughput and minimize I/O waits.
- To minimize I/O contention on individual disks.
- To support parallel I/O.
- To support parallel query processing, such as parallel star join or parallel table scan.
- To provide acceptable availability and recoverability from disk failures.

Designing a database allocation that meets these goals is a non-trivial task. Assigning database files to disks typically raises questions like: *Which tables or indexes should or should not share the same disks? How many of the available disks should be used for a given table or index?*

For example, you could argue that it is a good thing to avoid I/O contention between star index and fact table access by putting the fact table and the star indexes on different sets of disks. If 10 disks were available, you could use 5 disks for the fact table and 5 for the indexes. Although this configuration can reduce I/O contention, it also limits the degree of I/O parallelism for either fact table or star index access to 5.

Alternatively, you might argue that most of the I/O access will be random (because of non-sequential access of star index and fact table during a star join). Thus, mixing random I/O to tables and indexes on the same set of disks would not make much of a difference, and the difference would be even less for a multi-user workload. The consequence would be to distribute both fact table and star indexes over all 10 disks to maximize I/O parallelism to both.

What to do? Among experienced database experts there are advocates for either of these two approaches. And there are more choices to be made such as:

- Is it better to use disk striping or manual data file placement?
- What stripe unit size should be used?
- Where and how should you allocate temporary spill space?

The optimal solution for these and other questions really depends on your workload. However, it can be time-consuming to analyze workloads, and workloads can change over time. Even if you understand the query workload, translating the query workload into an I/O workload (that is, to sequences of physical I/Os to table and indexes) is very difficult and is dependent on dynamic

factors such as caching effects[1]. Therefore, data allocation based on workload analysis is theoretically interesting but in reality almost always infeasible [Ganger et al. 93].

In sum, it can be difficult to predict which of these two simple options would result in better I/O performance. The goal of this article is to discuss and determine *pragmatic* solutions for the data allocation problem in Red Brick. We focus on the problem of how to lay out and configure striped volumes (if any!), and how to assign Red Brick database files to the disks or logical volumes.

Assumptions:

- We assume that you have a farm of disks upon which you can create arbitrary striped volumes using storage managers such as Veritas Volume Manager or AIX® Volume Manager.
- We also assume that the access path to the disks is optimally configured and is not the bottleneck that needs tuning. Thus we do not focus on the configuration of disk controllers, fiber channels, etc., which connect the disk subsystem.

In this article:

- We review the allocation recommendations given by various storage and database vendors.
- Based on this, we propose several storage layout alternatives for a typical Red Brick data warehouse.
- We use extensive single-user and multi-user query performance tests to compare the advantages and disadvantages of these storage layout options.
- We describe the results of our performance measurements.
- We offer specific I/O configuration and tuning guidelines for IBM Red Brick Data Warehouse.

2 What the vendors say

Hardware and software vendors such as IBM, Oracle, Informix®, Sun, Compaq, EMC, Veritas have issued a variety of white papers and tuning recommendations. Some of them contradict each other and have to be adopted with care. For an excellent survey of disk striping, RAID technologies, and their application see [Chen et al. 94].

2.1 Vendor recommendations for storage layout

Recommendations for database storage layout come in a broad range of variety, often with specific qualifiers regarding the vendor's hardware or software product. For simplicity, we classify the storage layout guidelines to fall into one of two categories:

- Distribute all database objects over all disks.
- Separate database objects using disjoint (non-overlapping) sets of disks.

By database objects we mean tables, indexes, temp space, logs, database catalog/system tables, materialized views, and so on.

[1] Clearly, data caching has a significant impact on Red Brick performance and there are multiple levels of caching between the Red Brick engine and the disk platter, e.g. the database block cache, the OS file cache, controller cache, and often an on-disk cache for 1 track. The further down you look at the caching hierarchy the more difficult it is for the DBA or system administrator to determine the cache hit ratio or quantify its impact on overall system performance. Obviously, the higher the combined hit ratio of the caching hierarchy the lower the impact of how the data is organized on disk. So, with very large caches and a relatively low I/O load, different storage layouts make less of a difference.

Distribute "everything" over all disks

The basic idea is to spread all database objects as evenly as possible over the entire available storage subsystem to maximize I/O parallelism without worrying about I/O contention when different database objects are accessed concurrently.

For example, IBM has conducted a set of performance experiments to compare storage layout options for DB2® Universal Database™ on the IBM Enterprise Storage System (ESS)®, also referred to as *Shark* [IBM 01]. A key result is the recommendation to spread all DB2 data across as many RAID arrays as possible. It is suggested to intermix table, index, and tempspace across the disk arrays of the ESS to avoid the skewed and unbalanced I/O load that would occur if these were isolated. [IBM 01] presents specific tests that show that separation of table from index storage results in sub-optimal performance.

Another example is [Loaiza 00], who has coined the term *SAME* (Stripe And Mirror Everything), which suggests that all database objects should be *striped* over all available disks using 1MB chunks, with mirroring for reliability. He argues that today's database systems' workloads are too complex and have widely varying degrees of concurrency to enable one to design a storage configuration based on workload characteristics. *SAME* is workload-independent and intends to spread the I/O load evenly over all disks.

Because a database workload may focus intense I/O activity on any subset of the database (a table, an index, or a partition of a table), any such subset of the database should be equally spread over as many disks as possible ("extreme striping"). The number of disks should be as least 8 to 10 times the number of CPUs. *SAME* recommends not trying to separate index from table I/O, nor to separate sequential from random access, not even to place the recovery log on separate disks. The rationale is that the interference between index and table access or between log writes and table I/O is more than compensated for by the high bandwidth and reduced queueing provided by extreme striping. [Loaiza 00] argues that in practice striping over too few disks is a bigger problem than any I/O interference, and that I/O requests in a multi-user workload will interfere with each other in any case. Finally, he argues that *SAME* greatly reduces the storage administration overhead, which is also noted in [Larson 00]

Other vendors largely agree with these guidelines [Larson 00], [EMC 00]. Although data files can be distributed evenly across disks manually, this can very time-consuming and introduces risk for performance degradation due to datafile skew [Ganger et al. 93]. Instead, users are advised to *stripe* data, indexes, and temp space across all available disks. Note that advanced storage subsystems such as IBM's ESS or EMC's Symmetrix may require additional tuning considerations beyond the scope of this article.

Separate database objects using disjoint sets of disks

A common recommendation is to place the recovery log on disks that are separate from the table and indexes, not only for performance but also for fault tolerance. Other "separation" rules suggest separating tables from indexes or separating tables from each other if they are joined frequently. Yet another guideline found is separation of sequential from random access data [Compaq 98], [Whalen, Schoeb 99]. For example, [Holdsworth 96] advises that tables that are subject to multiple concurrent parallel scans be given a dedicated set of disks. While this may be beneficial for recovery logs, it is probably difficult for tables; i.e. [Larson 00] argues that a table scan running concurrently with an indexed lookup on the same table is unlikely to enjoy sequential I/O benefits.

Also, most database systems, including Red Brick and DB2, try to avoid full scans of large tables anyway.

For IBM Red Brick Data Warehouse [Fung 98] advocates separation of data objects and gives specific recommendations for configuring striped volumes.

- Have several striped volumes, all on disjoint sets of disks to separate fact table data, fact table indexes, dimension tables, dimension table indexes, and spill space.
- Ideally, a physical disk should hold data of only one table or one index; this is recommended for fact tables and their indexes but less important for small dimension tables.
- Do not use one large striped volume over all disks.
- Use a small stripe unit size (8 KB or 16 KB); larger chunk sizes would hurt performance.
- Each logical volume should span multiple controllers.

The recommendations in this category are based on the assumption that avoiding contention between different tables and between table and index access is more important than maximizing physical I/O parallelism to all database files alike. While this may be valid in special cases, measurement results discussed later in this paper show that this is not true in general.

2.2 Vendor recommendations for stripe unit size

[Holdsworth 96] notes that stripe unit size is a hotly debated issue. He argues that the stripe size should *not* be set to the database system's block size because it has been shown to degrade performance (in Oracle systems). He recommends a stripe size significantly larger then the database block size such as 1MB, and on very large systems the stripe size should be 5 MB. [Veritas 01] advises that the stripe size should be a multiple of the OS (or logical volume) block size. For most OLTP databases, Veritas recommends their default stripe unit size of 64 KB while Decision Support Systems (DSS) may perform better with larger stripe sizes, such as 128 KB or 256 KB. [Loaiza 00] argues that a 1 MB stripe unit size is suitable for any kind of workload because it leads to a good ratio of transfer times to seek times. He indicates that this value will increase in future as disk transfer rates increase faster than seek times decrease. For Red Brick, [Fung 98] recommends an 8 KB stripe unit size.

2.3 Research findings – stripe unit size

[Scheuermann et al. 98] is one of the later studies that unifies and confirms existing research. Their performance model is based on an open queueing network, which seems appropriate for today's database applications (many users, varying degree of concurrency). The results show that a small stripe unit size can provide good response times for a light load, but can severely limit throughput. Consequently, with small stripe units the response times increase drastically under heavy loads. A larger stripe unit size tends to cluster portions of a file on disk. While this may or may not improve the performance of individual I/O requests (depending on the request size), it is shown to increase overall I/O throughput if the requests are still uniformly distributed over all disks. Since a large stripe unit size, e.g. 64KB or 1MB, is still very small compared to a total database size, this uniformity assumption seems reasonable. Thus, theoretical results indicate that large stripe unit sizes provide better performance under medium to heavy workloads, which in database systems can arise due to multi-user activity, intra-query parallelism, or both.

3 Storage layout performance comparison

If you've made it this far, you can see that there are various recommendations for the storage layout of data warehouses such as:

- Manual datafile placement vs. disk striping.
- Separating indexes and tables on disjoint sets of disks vs. "extreme striping".
- Choosing a stripe unit anywhere between 8KB and 1MB, or even larger.

So we decided to conduct a set of performance experiments in Red Brick Data Warehouse 6.11 to test these recommendations and to see if we could come up with practical recommendations.

3.1 Data warehouse schema

In our performance tests, the database schema is a typical star schema with one large fact table (*daily_sales*) and 5 dimension tables. The dimension tables and their cardinalities are *period* (2522), *store* (63), *product* (19450), *promotion* (35), and *customer* (1,000,000). Each entry in the period table corresponds to one calendar day, but only 638 days are referenced in the fact table, which holds approximately 295 million rows. Fully loaded, the total database size is approximately 50GB. Even though this is small compared to many real world data warehouses, we found that it is big enough to perceive and analyze the performance impact of different storage layouts.

The fact table is indexed with two star indexes: one index includes 4 dimension foreign keys, and another includes 5. Each dimension table has a primary key index and typically one or more secondary indexes. The fact table and the two star indexes are range partitioned ("segmented") by value of the period foreign key. One segment holds one week's worth of data; that is, it spans 7 period keys. Thus, the fact table and both star indexes are comprised of 91 segments each. Each fact table segment and each star index segment is stored in 22 data files (PSUs).

3.2 Storage layout alternatives

The storage layout options we designed for the data warehouse focus on four main parts of the database:

- The fact table
- The fact table indexes
- The dimension table and indexes
- The spill space (temporary space for sorts, index building, etc.)

Our storage subsystem consists of 22 disks. We define seven different ways of allocating and distributing the database objects on the disks as follows (see Figure 1 for an illustration).

1. Manual 22

In this layout, we do not use any striping but place the PSUs manually on the 22 disks. For the fact table and star index segments, which are stored in 22 PSUs each, this means that each segment is distributed over all disks. The rationale for this is that even queries that access only a single segment can enjoy the maximum I/O parallelism of the disk subsystem.

One option is to simply allocate PSUs 1 to 22 of each fact table and index segment on disk 1 to 22 such that PSU *i* of each data and index segment would be stored on disk *i*. However, in real world installations, DBAs and consultants seem to favor a "1-off" strategy, such as starting with PSU 1 of data segment 1 on disk 2 and PSU 1 of the corresponding star index segments on disk 1, and then moving the pairings down the disks and at the end wrapping around to disk 1. This is a "1-off" logic between a fact table segment and its respective index segments.

Additionally one can apply a "1-off" logic between the fact table segments; that is, if the PSUs of data segment 1 are placed on disks 2, 3, 4, etc., then the PSUs of data segment 2 would be placed on disks 3, 4, 5,and so forth. The goal of the "1-off" logic is to reduce I/O contention between index and table access and between accesses to multiple fact table segments.

We use these "1-off" strategies in Manual 22. However, we did no specific measurements to verify or quantify the advantage of the "1-off" logic over a simpler manual distribution of files.

Furthermore, we create spill space directories on all of the 22 disks. Since it does not make sense to partition the small dimension tables into 22 pieces, we simply place each dimension table and each dimension index on a different disk. Most likely they are fully cached anyway. An exception is the *customer* table and its primary key index, which we store in 10 PSUs on 10 different disks each.

2. Manual 14/8

With Manual 14/8, we separate fact table data from fact table indexes on disjoint sets of disks. Using similar "1-off" strategies as above, we manually distribute fact table segments/PSUs over disks 1 through 14, and we distribute star index segments/PSUs over the remaining 8 disks, 15 through 22. For simplicity (and since it didn't seem to have much of an impact), we allocate spill space and dimensions like we did in Manual 22. The decision to use 14 disks for the fact table and only 8 disks for the star indexes is merely an intuitive one based on the fact that the star indexes are smaller than the table.

3. Stripe 12/4/4/2 (64 KB)

This layout follows recommendations in [Fung 98]; that is, it separates fact table, star indexes, spill space, and dimensions into different striped volumes on disjoint sets of disks. The numbers of disks in the striped volumes is again based on the relative size of the objects. The fact table is striped over 12 disks, the star indexes and the spill space over 4 disks each, and the dimensions over 2 disks. We chose a medium stripe unit size of 64 KB, which is the recommended default value in the Veritas Volume Manager that we used.

Fact table	12 disks (disks 1 through 12)
Star indexes	4 disks (disks 13 through 16)
Dimension and system tables	2 disks (disks 17 and 18)
Spill space	4 disks (disks 19 through 22)

4. Stripe 8/8/4/2 (64 KB)

Rethinking the 12/4/4/2 layout, one could argue that star index I/O would be as crucial for star join performance as the fact table I/O. Thus, the 8/8/4/2 storage layout is very similar to the 12/4/4/2 layout except that it assigns an equal number of disks to star indexes and the fact table (8 disks each) even though the fact table is bigger in size.

Fact table	8 disks (disks 1 through 8)
Star indexes	8 disks (disks 9 through 16)
Dimension and system tables	2 disks (disks 17 and 18)
Spill space	4 disks (disks 19 through 22)

5. Stripe 22 (64 KB)

This storage layout follows the concepts of "SAME" or "extreme striping" advocated in [Larson 00] and [Loaiza 00]. Here we use just one large striped volume over all 22 disks. All PSUs are stored in this volume. Again we use Veritas' default stripe unit size of 64 KB.

6. Stripe 22 (8 KB)

After our measurements indicated that extreme striping provides better performance and lower administration overhead than any of the other storage layouts, we decided to stick to this layout to investigate the impact of the stripe unit size. Following the recommendations in [Fung 98] we tried an 8 KB chunk size.

7. Stripe 22 (1 MB)

This is the same as 5 and 6 but with a 1MB stripe unit size, as recommended by [Holdsworth 96], [Loaiza 00].

Illustration of seven storage layout alternatives

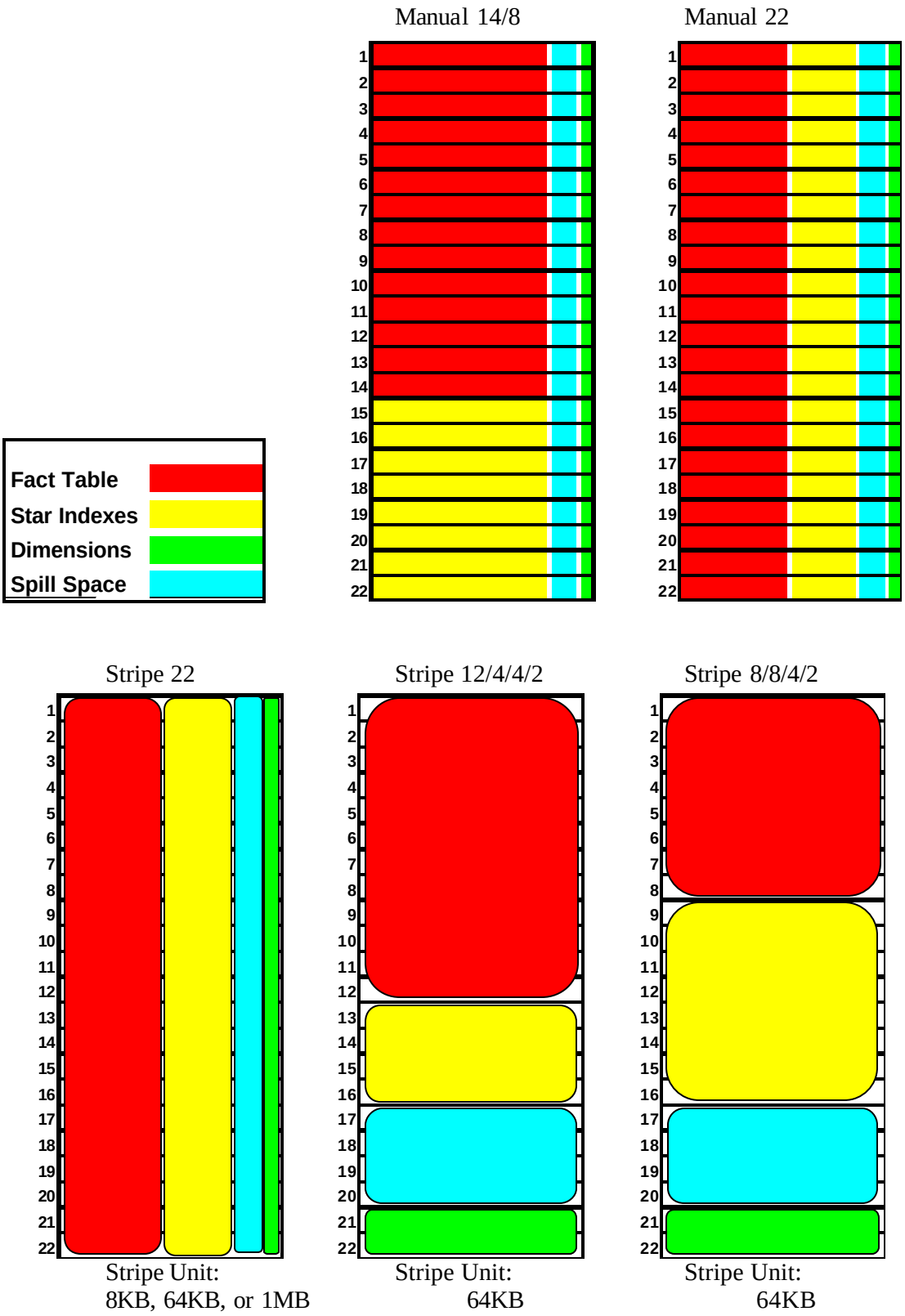

Figure 1: Storage Layout Alternatives

3.3 Workloads

We measured the performance of five different workloads for all of the storage layout alternatives described above. These workloads are:

Workload 1: Fact table load
This workload loads approximately 295 million rows into the empty fact table. This process includes building two indexes and full referential integrity checking against the preloaded dimension tables. The index building makes significant use of the spill space on disk.

Workload 2: Single user, with tightly constrained star joins
This set of queries consists of star join queries, which are constrained on several or all of the dimensions. They all use either one of the star indexes, and the percentage of rows finally selected from the fact table is very low, often less than 1%. The queries were executed in single user mode, that is, one at a time.

Workload 3: Single user, with a mixed set of queries
This is an extended set of queries that has both tightly constrained star joins and moderately and weakly constrained star joins, which place a higher I/O burden on the system. Additionally, there are queries that are forced to evaluate a join between fact and dimension tables by first scanning the fact table and then probing into the dimensions one by one. There are also a few queries that simply scan a portion of the fact table with and without additional constraints. Some of the queries in this set perform large sorts and aggregations that spill to disk.

Workload 4: Multi- user, with tightly constrained star joins
This is the same as Workload 2, but the set of queries is executed in a batch by many concurrent users. We ran this test with 10, 25, and 50 concurrent users. Each simulated user runs the batch of queries multiple times and each user submits the queries in a different order.

Workload 5: Multi-user, with a mixed set of queries
This is the same as Workload 3, but the set of queries is executed in a batch by many concurrent users. We ran this test with 10 and 20 concurrent users. Each simulated user runs the batch of queries multiple times and each user submits the queries in a different order.

4 Measurement results

We executed and measured each workload multiple times for each of the storage layout alternatives to ensure reproducibility and consistency of the results. To isolate the performance impact of the storage layout alternatives and to prevent other performance issues from distorting the results, the database system was highly tuned for each of the five workloads. To a certain extent, we also repeated the experiments for different sets of database configuration parameters. For example, we varied the degree of parallelism per query, the amount of main memory used by Red Brick, etc. We also experimented with sorted vs. unsorted data in the fact table and with pre-allocated data files vs. data files that get extended during the load process. These variations confirmed that the results and conclusions, which we present below, are robust and not very sensitive to such configuration changes.

We ran our measurements on a Sun E6500 server with 16CPUs and 16 GB of main memory running on Solaris 5.8 (64bit). For the disk subsystem, we used a Sun A5200 with 22 disks managed by Veritas Volume Manager. Specific test runs with reduced parallelism, low query

memory limit, and a cold OS file system cache confirmed that our qualitative results and conclusions are not distorted by (or overly sensitive to) the capacity of the hardware.

4.1 Fact table load

In the load test, all of the seven storage layout alternatives performed about equally well. Figure 2 shows the relative elapsed time of the load as a percentage of the fastest load. The best load performance was measured in the Stripe 8/8/4/2 layout while the load took about 4% to 12% longer in the other layouts. Even though 10% is not an insignificant difference, we found it to be within the range of run-to-run variations. Our tuning experience with the Red Brick loader confirms that the details of the storage layout play a minor role in performance. This is because for any sufficiently parallel I/O layout, the loading procedure is typically CPU-bound. The main requirement is that the I/O activity needs to be distributed over sufficiently many disks to keep the loader CPU-bound. This is the case for all of our seven storage layouts. The conclusion is that from our loader tests we cannot favor any one storage layout over the others.

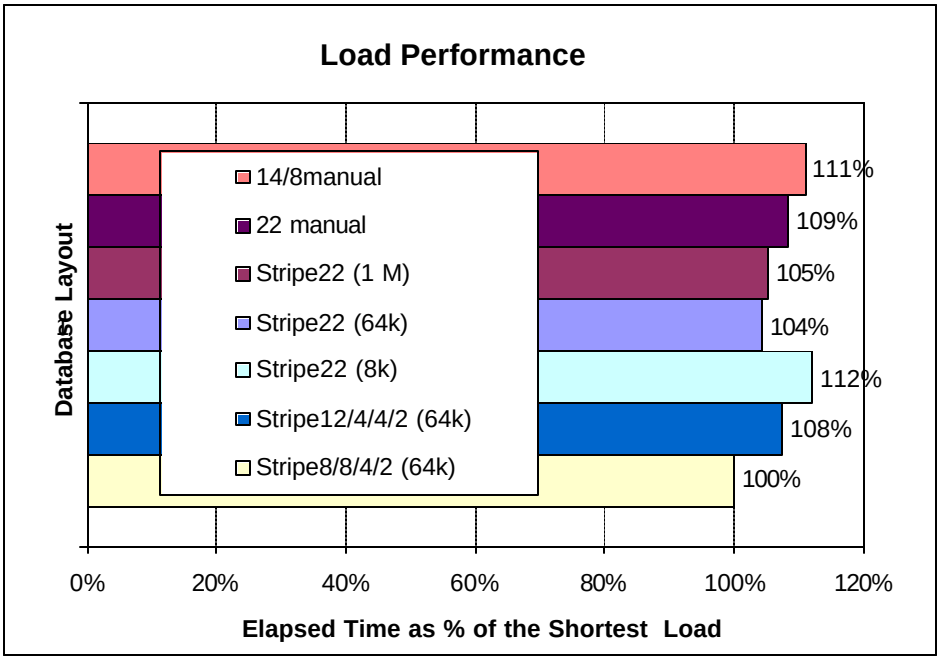

Figure 2: Load Performance for different storage layouts

4.2 Single user query workload

For each of the storage layout alternatives Figure 3 shows the total time to execute all of the tightly constrained star joins in a sequential manner. Because of Red Brick's optimized star join technology, tightly constrained star queries can have response times of a few seconds (or less) even on very large databases. Thus, for all of the storage layout options this workload took less than a minute to complete. We repeated this test many times with very low variance in run time and confirmed that the relative performance differences are statistically significant despite the short run time of the experiment. During the test we monitored the system activity and utilization and found that this workload places a light I/O load on the system, yet significant enough for the storage layouts to have an impact on performance. (Note that the storage layout may have no impact if a workload is heavily CPU-bound.)

Figure 3 shows that "extreme striping" with a small stripe unit size (Stripe 22(8KB)) performs best, closely followed by the manual data file allocation strategies. Because the I/O load is light and the degree of concurrency low, the performance advantage of the small stripe unit size over the larger ones is consistent with research results (see for example [Scheuermann et al. 98]). For this workload we find that the separation of indexes and tables does not make a significant difference; that is, Stripe 8/8/42, Stripe12/4/4/2, and Stripe 22 with a medium stripe unit size (64KB) all provide about equal performance. We believe that the I/O intensity in this workload is too low for reduced contention between index and table access vs. increased I/O bandwidth of extreme striping to be significant. This, however, is different for a mixed and more I/O-intensive workload (see discussion of Figure 4 later).

The result labeled "22 skewed" in Figure 3 was obtained in our first test series with 22 Manual. When distributing the data files evenly across the 22 disks, we paid close attention to implementing the "1-off" logic and to ensuring that all data files are similar in size. We found that this requires *orders of magnitude more skill and time* than simply striping everything across all disks. Sure enough, a little glitch in our estimation of the `maxsize` parameter for PSUs caused an uneven distribution of data over the 22 disks and resulted in a 35% performance degradation! Even though this happened by genuine accident, it teaches an important lesson: *striping is a much easier and much more reliable way of balancing I/O than placing the data manually*. (See [Ganger et al. 93].)

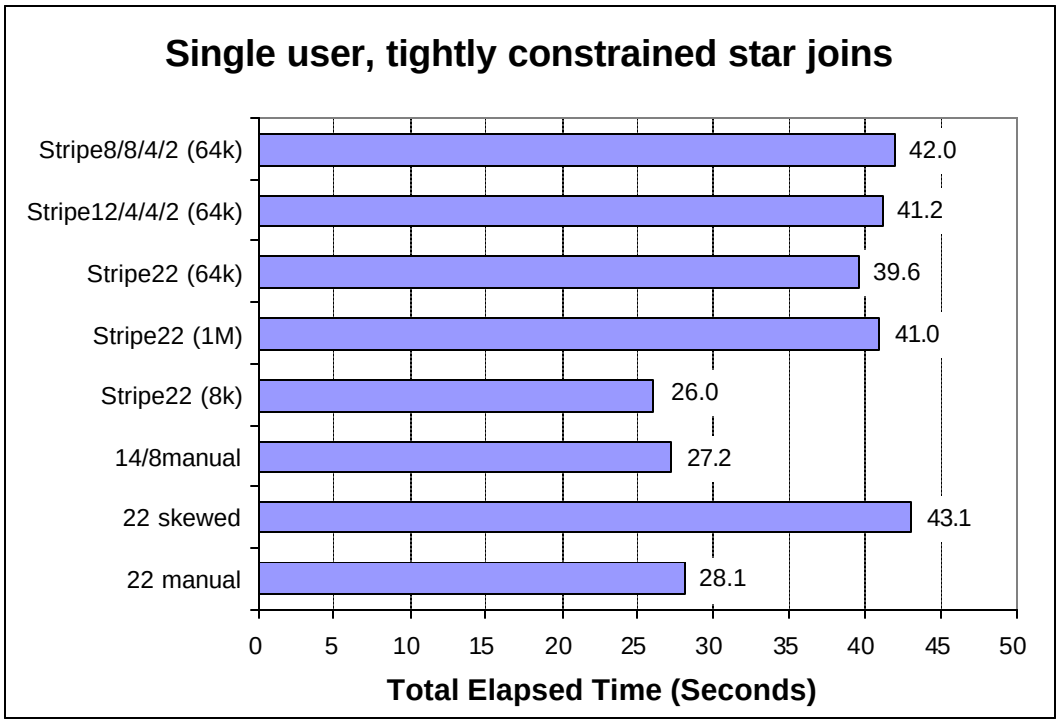

Figure 3: Total run times for a set of tightly constrained star join queries

Figure 4 shows the total time to sequentially run all of the mixed queries. These queries are both more CPU-intensive and more I/O-intensive than the previous workload. Thus, the total execution time of this workload was up to one hour. The results show clearly that striping everything over all disks with a medium to large stripe unit size provides best performance and maximum ease of administration. We find that separating fact table I/O from star index I/O in non-overlapping striped volumes does *not* offset the benefit of increased parallelism provided by extreme striping. It is

interesting however that the 12/4/4/2 layout performed better than the 8/8/4/2 layout. This means that the fact table I/O is more significant than the star index I/O and that devoting more I/O parallelism/bandwidth to the fact table is advantageous. Indeed, when we monitored the I/O utilization per logical volume during tests with the 8/8/4/2 layout we found that the fact table volume was "hotter" than the star index volume. (We mean "hotter" in terms of I/O wait time, queue length, etc.). This confirms the statements in [Holdsworth 96], [Larson 00], and [Loaiza 00] saying that separating indexes from tables can create I/O hotspots.

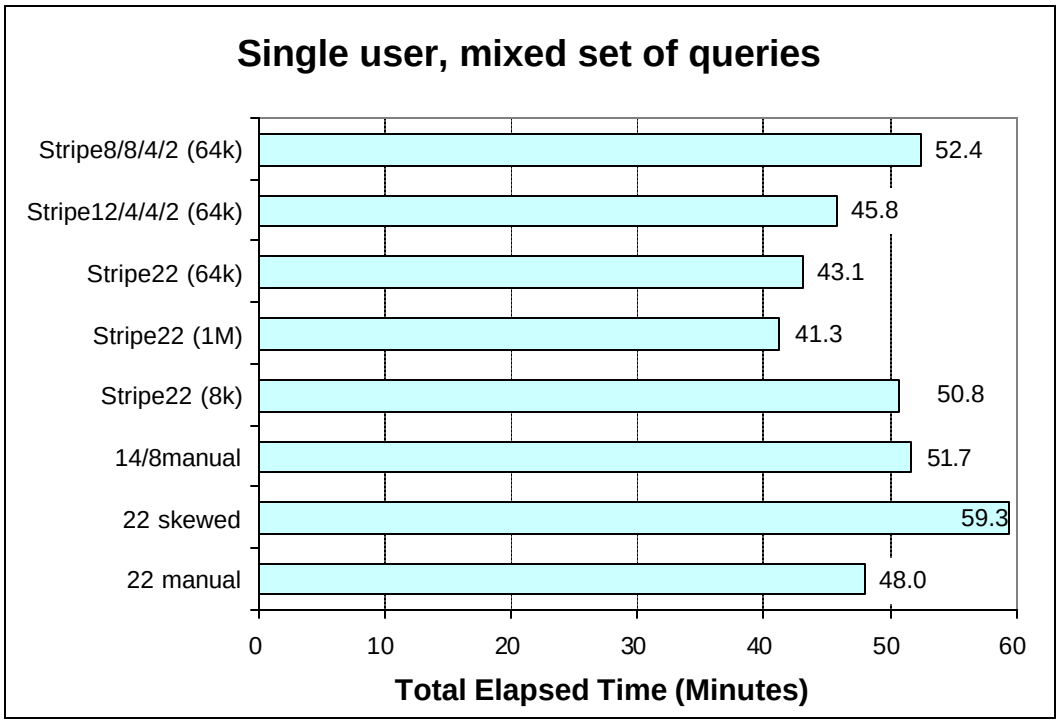

Figure 4: Total run times for a mixed set of queries

Because the result in Figure 4 is based on a blend of different types of queries (index-based star joins, table scans, sorts with spilling, etc.) you might wonder if the result is dominated by any single type of query. For example, does the result in Figure 4 look the way it looks only because there are table scans in the workload? The answer is "no". With the exception of tightly constrained star joins (Figure 3), every type of query in our workload has (on average) the relative performance characteristics shown in Figure 4. Certainly, there are individual queries that are exceptions. For example there is one table scan query in our workload that performs better on "22 manual" than on "Stripe22 (1MB)" but the other table scans perform better on "Stripe22 (1MB)". However, our goal is not to find the storage layout that optimizes the performance of one particular query, but to find one that optimizes the *overall performance of a representatively mixed workload*.

Having said that, one might wonder how a table scan on a striped volume can possibly compete with the sequential I/O of contiguously allocated data files in a layout like Manual 22. The key thing is that sequential I/O is not equal to reading an entire file without intermediate disk seeks. Sequential I/O is when the seek time is a sufficiently small percentage of the data transfer time. For example, 10 random reads of 10MB each are almost as efficient as a single scan of 100 MB. Likewise, with an increasing stripe unit size (64 KB, 512 KB, 1 MB) we can achieve an increasingly smaller ratio of seek time to continuous transfer time, which provides a sufficient approximation of sequential I/O.

If you compare the results of the two single user workloads in Figure 3 and Figure 4, note that Figure 3 is scaled in *seconds* while Figure 4 is scaled in *minutes*. In both cases extreme striping layout Stripe 22 provides best performance. For tightly constrained star joins, a small stripe unit size of 8 KB proves to be beneficial, while a mixed workload performs much better with a medium or large stripe unit size. But tightly constrained star joins perform reasonably well anyway (*seconds*!), while more I/O-intensive queries are much more dependent on the "right" storage layout. In many data warehouse installations, queries may often take many minutes rather than seconds to complete, typically when there are weak constraints, complex joins over many tables, and a very large data volume. Most customers would happily spend a few extra seconds on queries which are very fast anyway if it saves them many minutes of the long queries. Thus, unless a workload is known to consist predominantly of tightly constrained star joins, a medium to large stripe unit size of at least 64 KB is recommended.

4.3 Multi-user query workload

In a second phase of experiments we executed the two query workloads in a multi-user fashion. We ran the tightly constrained queries with up to 50 concurrent simulated users and the mixed set of queries with up to 20 users. Even though the absolute results are dependent on the number of users, the *relative* performance differences between the storage layouts were similar for different numbers of users. Thus, we present results for only one case per workload.

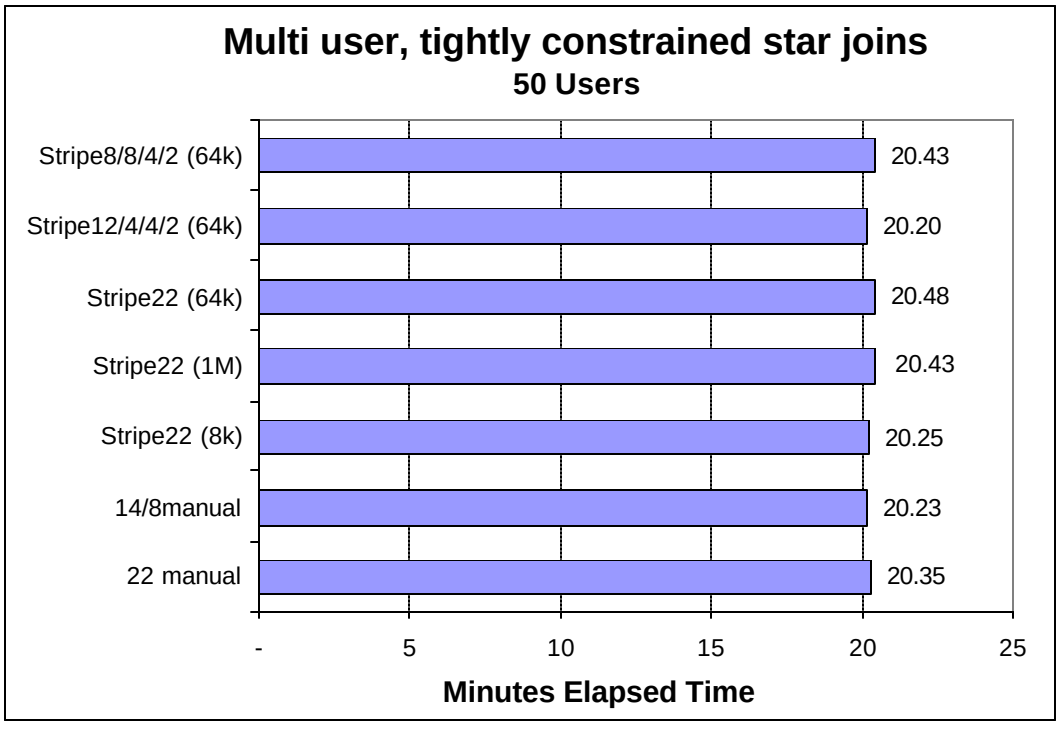

Figure 5: Total time for all 50 users to complete the workload

For each of the storage layout alternatives Figure 5 shows the total elapsed time for 50 concurrent users to run the set of tightly constrained star joins multiple times. None of the storage layouts performed significantly worse or better than any of the others. Interestingly, the performance advantage observed for Stripe 22 (8 KB) in the single user case does not translate to a corresponding performance benefit in the multi-user case. We believe that the increased degree of

concurrency defeats the performance benefit of the small stripe unit size that we saw in Figure 3. Yet, the overall I/O intensity is still low so that the performance advantage of extreme striping and a larger stripe unit size isn't sufficiently significant. Indeed, we observed that this concurrent workload saturates the CPUs much sooner than the disks.

The I/O intensity -- and thus the performance behavior – of the multi-user workload changes significantly as more moderately constrained queries are added to the mix. Figure 6 shows the total elapsed time of the multi-user test with the mixed set of queries being executed multiple times by every user. The I/O load on the system is medium to high with a significant degree of inter- and intra-query concurrency. This amplifies the qualitative results already seen in Figure 4. A very small stripe unit size is not suitable to satisfy the demand for high I/O transfer rates. With 8 KB stripes, the seek-time to transfer-time ratio is too high, which leads to excessive queueing. This results in very poor overall performance. Using a 64 KB or 1 MB stripe unit size greatly relieves this problem and performs much better. This is consistent with research results, which predict that a small stripe unit size severely limits throughput so that response times deteriorate under a heavy load due to queueing delays [Scheuermann et al. 98].

Figure 6 also confirms and amplifies the result that separating index from table I/O leads to unbalanced and thus sub-optimal disk utilization. Because fact table I/O is heavier than star index I/O, performance improves as more disks are devoted to the fact table, in the order of 8/8, 12/4, 14/8, and 22. A skillful manual distribution of data files over all disks (Manual 22) does not perform quite as good as Stripe 22 (1MB) or Stripe 22 (64KB), but performance is not disastrous either. This is good news for existing data warehouse installations that deployed this strategy. Manual 22 performs reasonably well because it does not separate different database objects onto disjoint sets of disks, does not use a very small stripe unit size, and provides some approximation to spreading the I/O load over all available disks. Nevertheless, the administration effort is large, and so is the danger of inadvertently introducing data skew and unbalanced I/O load, which can severely degrade performance. Both ease of administration and performance results recommend striping over manual data file placement.

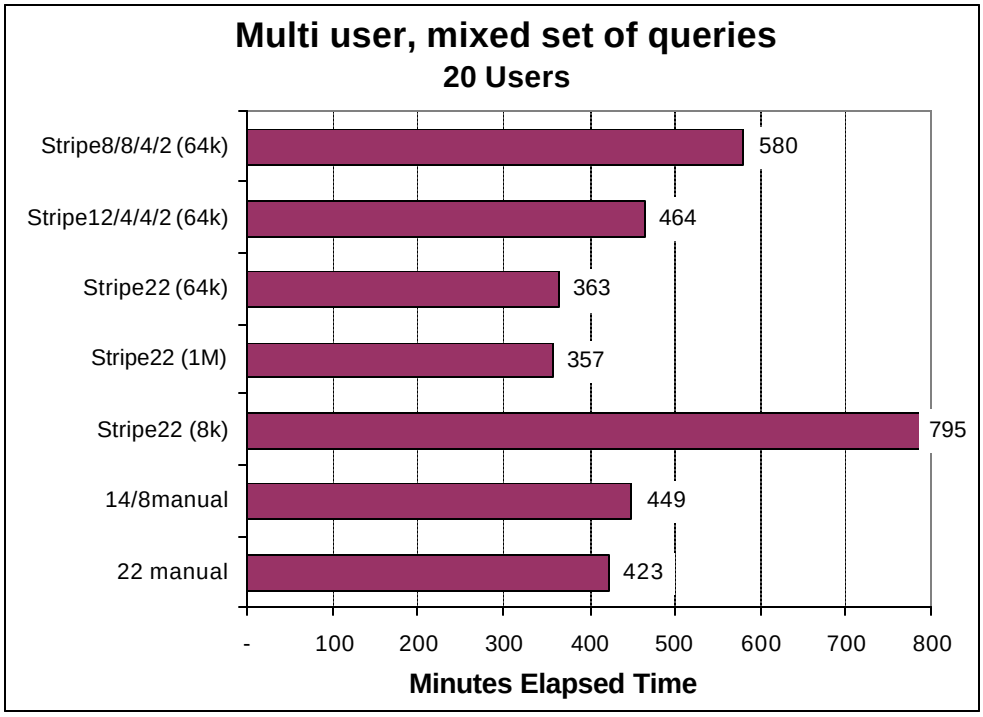

Figure 6: Total time for all 50 users running a mixed query workload

5 Conclusions and tuning recommendations

Our experiments show that distributing all Red Brick database files (and thus the I/O load) evenly over all available disks by means of striping provides very good performance at low administration overhead. This confirms the validity of storage layout strategies described as "extreme striping" or SAME (see Section 2) and is also consistent with research results. Specifically, the separation of tables from indexes is not required and can adversely affect performance.

Below we summarize the main tuning recommendations for IBM Red Brick Data Warehouse. These tuning guidelines cannot be expected to provide optimal performance for *every* possible query and application scenario. Instead, they are intended to optimize the overall performance of a *representatively mixed* workload. We emphasize that these are just rough guidelines and that a specific data warehouse installation or the characteristics of a particular computing environment may require adjusting these strategies. Thus, we do not encourage you to follow our recommendations blindly. We do hope that you come away from this article with a better understanding of the choices and potential consequences of different storage layouts. This in turn should help you make educated decisions when designing the storage layout for a particular data warehouse implementation.

5.1 I/O tuning Recommendations

- **Use a sufficiently fine granularity for the segmentation scheme**
  - Partition the fact table into a reasonable number of segments rather than having just a few huge segments. Hundreds of segments can be OK.

- Note that Red Brick star join and target join parallelize by segment. If most join queries access one month's worth of data, segmenting the fact table by month prevents default parallelism. In this case, it is better to partition by week, or maybe even by day.
- Note that various Red Brick operators can infer if certain segments do not need to be accessed (depending on the query). This is known as "smart-scan". Fine granularity segmentation is better suited for excluding as much data from query evaluation as possible.

- **Store segments in a sufficiently large number of data files (PSUs).**
  - It is better to have a large number of smaller PSUs than a just a few very large PSUs per segment. This recommendation holds for both table and index segments.
  - If a single segment becomes a temporary I/O hot spot, spread the I/O over the entire width of the disk farm for optimum performance.
  - Note that table scans parallelize by PSU. Having too few PSUs per segment hinders parallelism.
  - For large tables and a moderate number of very large disks, we recommend having as many PSUs per segment as there are disks in the striped volume on which the table is stored. This is particularly important if you choose manual file allocation over striping despite the performance and administration advantages of striping.
  - For larger numbers of disks (e.g. 64 or more) this may lead to too many too small PSUs. In this case have at least as many PSUs per segment as CPUs in the system. For fact table segments this should be the absolute minimum. If this results in very big PSUs (>500 MB or >1 GB), use 2 times or 3 times as many PSUs as CPUs in the system.

- **Stripe everything over all disks.**
  - Ideally, use one large striped volume across all available disks. Store all PSUs in that one large volume. This provides optimum I/O load balancing, very good I/O performance, and very low administration overhead. Ideally, all disks should be of the same type (same size, same speed, etc.).
  - Depending on the actual storage system and volume management software available, one striped volume across all disks may not be possible. For example, there may be more disks than the maximum number of devices allowed in a striped volume. In this case maybe you can create two striped volumes over half the number of disks each. Each segment that is stored in multiple PSUs should have PSUs 1,3,5, and so on, allocated in the first striped volume and PSUs 2,4,6, and so on, in the second striped volume.
  - If limits force you to have even more than 2 striped volumes on disjoint sets of disks, make sure that each volume spans the same number of disks. Then distribute the PSUs of each segment evenly across the volumes, possible deploying a "1-off" strategy as discussed in section 3.2.
  - As with PSUs, distribute spill space over all disks.
  - Consider fault tolerance as discussed in section 5.2.

- **Use a medium to large stripe unit size (at least 64 KB, up to 1 MB).**
  - Do not use a smaller stripe unit size, such as 8 KB, unless you have specific evidence that in your particular case this will provide better performance.

- **Set the TUNE FILE GROUP and TUNE GROUP parameters.**
  - Example: If a large striped volume over 96 disks is named `/data`, be sure to have entries `TUNE FILE GROUP 1 /data` and `TUNE GROUP 1 96` in the `rbw.config` file.

This defines `/data` as file group number one and sets 96 as the maximum number of parallel processes that can access that volume concurrently (per query).

- If you don't set TUNE GROUP you are stuck with one process per query that can do I/O on `/data` at a time.
- Actually, it is enough to set TUNE GROUP to the QUERYPROCS value to allow all processes of a query to do I/O (assuming QUERYPROCS is less-than or equal-to the number of disks). However, if QUERYPROCS gets increased and you forget to also increase TUNE GROUP, you may end up with sub-optimal I/O parallelism. Thus, it doesn't hurt to keep TUNE GROUP set to the number of disks.
- If QUERYPROCS is larger than the number of disks, you may have an unbalanced system configuration. Typically, a system should have 4 to 5 times as many disks as CPUs, or more. Setting QUERYPROCS to 5 times the number of CPUs would be considered very high, maybe too high as it might lead to excessive context switching.

5.2 Considerations for fault tolerance

The key disadvantage of striping all data files over all available disks is greatly reduced fault tolerance. A single disk failure might make all PSUs unavailable. The probability that one disk out of *n* fails increases significantly with *n*. Availability concepts such as mirroring (RAID 0+1) and data parity (RAID 5) can be used to reduce this risk. Both RAID5 (parity) and RAID1 (mirrors) offer fault-tolerant disk storage and have their advantages and disadvantages.

The most fault-tolerant and best performing protection against disk failure is mirroring (RAID 0+1). Modern mirroring solutions support parallel writes to a disk and its mirror as well as balancing read request evenly between two disks of a mirrored pair. Mirroring requires two times the projected number of disk drives, which might be a cost issue.

RAID5 disk arrays are a more cost efficient form of fault tolerance, albeit at the expense of lower write performance. RAID5 disk arrays perform multiple physical I/Os for every logical write. Additionally, software-based parity computations may impose a significant overhead on write-intensive database operations (load, updates, etc.). The overhead of hardware-based parity computation (XOR chip on the RAID controller) is said to be negligible. Note also that the fault tolerance of RAID5 is lower than that of RAID 0+1. A RAID5 array can tolerate only one disk failure while RAID0+1 can tolerate multiple disk failures as long as not both disks of a mirroring pair go down.

Usually the number of disks in a RAID5 disk array is much more limited than in a RAID 0 or RAID0+1 volume (e.g. 7 or 14 disks only). As a possible scenario, assume that for fault tolerance 70 available disks need to be configured as 10 RAID5 arrays with 7 disks each. We could choose a 64KB stripe unit size for each of the 10 RAID5 arrays. For ease of administration and to evenly distribute the PSUs over the 10 RAID5 volumes, we would suggest defining a logical striped volume across the 10 RAID5 disk arrays. Each of the 10 RAID5 arrays looks like a single device to the logical volume manager. The stripe unit size of the logical volume could be set to 384KB, which equals 6 times 64KB. This means that one *logical* stripe unit would be written to 6 disks of a single RAID5 array with the parity block being written to the seventh disk. This leads to very good and most balanced utilization of the RAID5 arrays. Such two-dimensional striping (software striping across hardware striped volumes) is similar to *Plaids* in EMC systems.

About the author:
Matthias Nicola received his Diploma and his Doctorate degree in Computer Science from the Technical University of Aachen (RWTH) in Germany in 1995 and 1999 respectively. He worked as a researcher at the department of information systems at RWTH Aachen from 1995 to 1999. His research focused on the performance evaluation of distributed and replicated database systems. In 2000 Matthias joined Informix Software in California and worked for over two years on Red Brick Data Warehouse performance. Currently, he is working at IBM Silicon Valley Lab on DB2 and XML performance.
http://www.matthiasnicola.de/

References

[Bellatreche 00] Ladjel Bellatreche, Kamalakar Karlapalem, Mukesh K. Mohania, M. Schneider: *What Can Partitioning Do for Your Data Warehouses and Data Marts?*, International Database Engineering and Application Symposium, pages 437-446, 2000.

[Chen et al. 94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson: RAID: *High Performance, Reliable Secondary Storage*, ACM Computing Surveys, Vol.26, No.2, pp.145-185. June 1994.

[Chen, Towsley 96] Shenze Chen, Don Towsley: *A Performance Evaluation of RAID Architectures*, IEEE Transactions on Computers, Vol. 45, No. 10, pp. 1116-1130,October 1996.

[Compaq 98] *Configuring Compaq RAID Technology for Database Servers*, Compaq White Paper, May 1998.

[EMC 00] EMC: *Metavolumes and Striping*, EMC White Paper, 2000.

[Fung 98] Cindy Fung: *Disk Striping (Red Brick Warehouse 5.1)*, Technical Note TN9801, Red Brick.

[Ganger et al. 93] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt: *Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement*, Proceedings of the 26th International Conference on System Sciences, Vol. 1, pp. 40-49, 1993.

[Griffith et al. 99] Nigel Griffith, J. Chandler, J. Marcos, G. Mueller, D. Gfroere: *Database Performance on AIX in DB2 UDB and Oracle Environments*, IBM Redbook, December 1999.

[Holdsworth 96] Andrew Holdsworth: *Data Warehouse Performance Management Techniques*, Oracle Services White Paper, 1996.

[Larson 00] Bob Larson: *Wide Thin Disk Striping*, Sun Micorsystems, White Paper, October 2000.

[IBM 01] Barry Mellish, John Aschoff, Bryan Cox, Dawn Seymour: *IBM ESS and IBM DB2 UDB Working Together*, IBM Red Book, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246262.pdf, October 2001.

[Loaiza 00] Juan Loaiza: *Optimal Storage Configuration Made Easy*, Oracle Corporation White Paper.

[Scheuermann et al. 98] Peter Scheuermann, Gerhard Weikum, Peter Zabback: *Data partitioning and load balancing in parallel disk systems*, VLDB Journal, (7) 48-66, 1998.

[Veritas 01] *Configuring and Tuning Oracle Storage with VERITAS Database Edition ™ for Oracle*, VERITAS Software Corporation, White Paper, 2001.

[Whalen, Schoeb 99] Edward Whalen, Leah Schoeb: *Using RAID Technology in Database Environments*, Dell Computer Corporation Magazine, Issue 3, 1999.