Overview of Performance Guidelines

Logical Optimization
Performance of the database begins when the logical structure is designed. Normalization of tables, database page size, and indices are critical components of tuning.

Physical Optimization
Database tables by default should be normalized to 3rd Normal Form. There are instances, especially when dealing with mass amounts of data, which de-normalizing the table will provide greater performance because all the data are contained in a single table. Query design will be covered later, but it should be obvious that it is much faster to access a single table versus multiple tables.

Analysis Tools
Using the InterBase SQL (ISQL) tool we can define and/or extract the current physical definition of the database. The easiest way to define or change metadata is to execute a text file containing the SQL DDL statements with the **File -> Run an ISQL script** menu options. We can either extract the current definition a database to a file using the **Extract -> SQL Metadata for Database**. Or we can view the individual pieces using the View dialog box.
With the InterBase Server Manager tool we can examine how the physical definition of the database affects storage of data within the database file. Select **Tasks -> Database Statistics** to pop-up a new window and select **View -> Database Analysis** to see the number of pages allocated and their fill percentages.

Query Optimizer
How the Optimizer Works
The purpose of the optimizer is to select the most inexpensive method of retrieving information requested in a query based on the current state of the database. InterBase performs the optimization for each query the first time that query executes for the database connection. If the application detaches and reattaches to the database then all the queries for that database are optimized when they are executed again. The benefits of this method are that the queries will be optimized based on the current state of the database, i.e., how many and what types of indices are defined, how useful those indices are, and approximately how many rows in each table.
The optimization strategy or plan, is a series, from left to right, of retrievals from each individual table with the most expensive being the outer most retrieval and the most inexpensive being the inner most retrieval. To help clarify how this is done, we will use the following query on the employee.gdb database as an example:

```
SELECT * FROM employee e, department d, job j
WHERE e.dept_no = d.dept_no
    AND e.job_code = j.job_code
    AND e.job_country = j.job_country
    AND e.job_grade = j.job_grade
    AND e.dept_no < 120
```

- Retrieve from employee
- Retrieve from jobs
- Retrieve from department

The optimizer performs these seven major steps:

Decompose the boolean expression into a set of conjuncts or associations.
Distribute the equalities and inequalities.
Enumerate those conjuncts that can use an index to create a
Enumerate all possible permutations of indexed joins from the streams above into "".
Select the longest river.
Remove the selected river from the list and until all rivers have been selected.
Finally, take the set of rivers selected and try to generate sort merge joins between them; otherwise, do a cross product.

To generate the following plan:

```
PLAN JOIN (E INDEX (RDB$FOREIGN8), J INDEX (RDB$PRIMARY2, MINSALX, MAXSALX, RDB$FOREIGN3), D
INDEX (RDB$PRIMARY5))
```

1. Decompose

Decompose the boolean expression into a set of conjuncts or associations. Boolean expressions being equalities, inequalities, etc. From the example, we would end with the following conjuncts:

```
e.dept_no = d.dept_no
e.dept_no < 120
e.job_code = j.job_code
e.job_country = j.job_country
```

2. Distribute

Distribute the equalities and inequalities. If a=b and a=c then b=c is added to the set of conjuncts. The same applies for a constant, e.g., if a=5 and a=c then c=5 is added to the set of conjuncts. From the example, we would only create one new conjunct:

```
d.dept_no < 120
```

3. Create Streams

Enumerate those conjuncts that can use an index to create a stream, which is a set of records, such as a relation or a relation with restrictions on it. From the example, we would create the list of streams from the indices defined for employee, department, and job:

Indices:

Employee:

- `RDB$FOREIGN8:` duplicate on field dept_no
- `RDB$FOREIGN9:` duplicate on field job_code, job_grade, job_country

Job:

- `RDB$PRIMARY2:` unique on field emp_no
- `MINSALX:` duplicate on field job_country, min_salary
- `MAXSALX:` duplicate on field job_country, max_salary
- `RDB$FOREIGN3:` duplicate on field job_country

Department:

- `RDB$PRIMARY5:` unique on field dept_no

Streams:

- Employee with index `RDB$FOREIGN8`
- Employee with index `RDB$FOREIGN9`
- Job with index `RDB$PRIMARY2`
- Job with index `MINSALX`
- Job with index `MAXSALX`
- Job with index `RDB$FOREIGN3`
- Department with index `RDB$PRIMARY5`

4. Create Rivers

Enumerate all possible permutations of indexed joins from the streams above into "rivers". A "river" is a combination of multiple streams joined over the indexed fields.

From the example, we would create these permutations:

1. Employee to Jobs over job_code, job_grade, job_country
2. Employee to Department over dept_no
3. Employee to Job over job_code, job_grade, job_country to Department over dept_no
4. Employee to Department over dept_no to Job over job_code, job_grade, job_country

5. Select River

Select the longest river (assumed to be the cheapest); if two rivers are of equal length, then do a cost estimate to select the cheaper river. The cost estimate depends on several variables: whether there is an index, the selectivity (estimate of usefulness) of that index, whether there are selection criteria, the cardinality (approximate number of values) of the underlying relation, and whether the stream needs to be sorted. For example, when retrieving an indexed equality, the cost is estimated as:

*cardinality of base relation * selectivity of index + overhead of index retrieval*

The optimizer approximates the cardinality to be equal to:

the number of data pages for the relation / maximum number of records per page

Selectivity is equal to the estimated number of distinct values divided by the cardinality. The cardinality can change over time when new pages are allocated and records are stored, modified, or deleted. The selectivity is only set on index creation. There is a command that will recompute the selectivity for non-unique indices:

```
SET STATISTICS INDEX RDB$FOREIGN8;
```

From the example, there are two rivers with the same length, rivers three and four. Because they are the same length, we need to compute the cost associated with each river. River three is chosen because doing an index lookup on Job will be the least expensive retrieval as compared to Department. That is because there are only fourteen unique job_codes compared to twenty-four unique departments. Note that the selectivity of a unique index is "perfect", that is, looking up a value will find only one record with that value. A lookup into Employee is the most expensive because it uses a duplicate index and there are fifty- nine duplicate departments.

6. Repeat

Remove the selected river from the list generated in step 4 and repeat step 5 again. Repeat until all rivers have been selected.

7. Merge Rivers

Take the set of rivers selected and try to generate sort merge joins between them; otherwise, do a cross product. A sort merge means that both streams are sorted and the results merged. With the streams sorted, the database engine can scan through both streams just once to form the join. Otherwise, it must iterate through the second stream for each record in the first stream. A cross product means that every record in the first stream joins with every record in the second stream.

This should make it very clear that properly designed indices are a major influence on performance. What may not be so clear is that the selectivity of an index is only computed for a duplicate index. If the values in the index are changed, i.e. updated, deleted, new ones inserted, it is advised to recompute the selectivity periodically. This would affect only those applications that attach to the database after recomputing. Any existing queries will still be optimized using the old selectivity.

Database pages allocated for the table also impact the cardinality. If you load a great deal of data into a relation, delete a substantial number of rows, then you may have many pages that are empty or nearly empty. This will significantly affect the computation of the cardinality. The only method to correct this is to backup and restore the database.

So in design queries, it is very important that the primary and foreign keys have an index defined because these are the fields that tables will be joined across. The program *ISQL* is an excellent environment for prototyping queries. Make sure that the data returned are correct. Partial Cartesian products are much more difficult to detect then full ones.

Indices

InterBase index management is different from other Relational Database vendors. InterBase can use indices as both *navigational* or *bit-mapped*. **Navigational** means that the index is stepped through value by value in whatever order it was defined. This allows a query to access the first record very quickly if there is an index defined that matches the ORDER BY clause of the query. **Bit-mapped** means that multiple indices can be used to match the JOIN and/or ORDER BY clauses and that each index is scanned for matching values before all the indices are combined using an boolean OR operation.

Multiple Indices

Indices are probably the most important part of the tuning your InterBase database. As you can see InterBase depends heavily on indices for query optimization. No indices are defined by default, but with SQL's **Declarative Referential Integrity** syntax, the SQL CREATE TABLE will create unique indices for the primary key fields and duplicate indices for the foreign key fields.

```
CREATE TABLE foo (
    foo_num INTEGER,
    foo2_num INTEGER,
    PRIMARY KEY (foo_num),
    FOREIGN KEY (foo2_num) REFERENCES foo2 (foo2_num));
```

Because of this you must be very careful in defining your own indices as InterBase will use multiple indices to resolve queries. This can result in a degradation of performance if multiple indices reference the same fields as needed by the query. An example would be if the primary key of table *first_table* is made of three fields, *field_1*, *field_2*, and *field_3*. *Field_2* is a foreign key into table *second_table* and *field_3* is a foreign key into table *third_table*. Using the Declarative Referential Integrity syntax above, the following indices would be created:

```
RDB$PRIMARY1 UNIQUE ASC  (field_1, field_2, field_3);
RDB$FOREIGN1 ASC (field_2);
RDB$FOREIGN2 ASC (field_3);
```

Now, whenever you queried *first_table* and supplied values for all three fields, all three indices would be used when you only really need the unique index. There is an alternative if you really need to access fields *field_2* and *field_3* independently of *field_1* and you want to enforce uniqueness for *first_table*. The alternative is to drop the unique index on the three fields and create an artificially generated unique id field, *field_X*, with a unique index on that field and a duplicate index on field *field_1*. A trigger would have to be defined to generate the value for the id field *field_X*.

```
CREATE TABLE first_table (
    field_X INTEGER,
    field_1 INTEGER,
    field_2 INTEGER,
    field_3 INTEGER,
```

```
    PRIMARY KEY (field_X),
    FOREIGN KEY (field_2) REFERENCES second_table (field_2),
    FOREIGN KEY (field_3) REFERENCES third_table (field_3));

CREATE ASC INDEX first_table_fx1 ON first_table (field_1);

CREATE GENERATOR first_table_id;

SET TERM !! ;
CREATE TRIGGER first_table_unique_id FOR first_table
BEFORE INSERT
POSITION 0 AS
BEGIN
    new.field_X = gen_id ( first_table_id, 1 );
END;
SET TERM ;!!
```

Server Manager Statistics

As we discussed previously, the InterBase Server Manager (or the GSTAT console utility) can retrieve database statistics as they relate to the physical layout of the database file(s). The output from the Database Statistics option will look similar to this:

```
EMPLOYEE (34)
    Primary pointer page: 251, Index root page: 252
    Data pages: 5, data page slots: 5, average fill: 68%
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 2
        80 - 99% = 2

Index NAMEX (1)
    Depth: 1, leaf buckets: 1, nodes: 42
    Average data length: 15.00, total dup: 0, max dup: 0
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 0
        80 - 99% = 1

Index RDB$FOREIGN8 (2)
    Depth: 1, leaf buckets: 1, nodes: 42
    Average data length: 0.00, total dup: 23, max dup: 4
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 0
        80 - 99% = 0

Index RDB$FOREIGN9 (3)
    Depth: 1, leaf buckets: 1, nodes: 42
    Average data length: 6.00, total dup: 15, max dup: 4
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 1
        60 - 79% = 0
        80 - 99% = 0

Index RDB$PRIMARY7 (0)
Depth: 1, leaf buckets: 1, nodes: 42
Average data length: 1.00, total dup: 0, max dup: 0
Fill distribution:
     0 - 19% = 0
    20 - 39% = 1
    40 - 59% = 0
    60 - 79% = 0
    80 - 99% = 0
```

The text is information on tables and indices sorted alphabetically by table name. The most interesting information on a table is the number of **data pages** and the **average fill**. The information is usually only relevant for one table. This table should be the main table for the database, one that is read and/or updated constantly. If the *average fill* is below 60 percent then try backing up and restoring the database. If the *average fill* is still low then it might be advisable to increase the database *page size* to the next value.
The information on **indices** is more complicated. Essentially the only field you do not need to check is *leaf buckets*. InterBase uses a variant of B-tree indices and the field *depth* refers to the depth of the tree; the number of levels down from the top bucket. Normally this value should be three or less. If it is greater than three, the indices should be rebuilt. The command to do this in *ISQL* is:

```
ALTER INDEX custnamex INACTIVE;
ALTER INDEX custnamex ACTIVE;
```

If the *depth* does not decrease then it might be advisable to increase the database *page size* to the next value.
NOTE: The only way to rebuild the indices defined with the Declarative Referential Integrity syntax is to backup and restore the database.
The *nodes* is the total number of data values in the index. *Total dup* is the number of duplicate values and *max dup* is the largest number of duplicates for a single value. *Average data length* is *the total number of compressed bytes for all data values / nodes*. The data values are both pre and postfix compressed. Postfix uses run-length encoding. Prefix compression is best explained using an example. Using the index *custnamex* which is an ascending index on the field *customer*, a text field of a maximum twenty-five characters. Given the data values of:

```
AA Auto Rentals
AAA - California
AAB Incorporated
AAB Towing
ABC Limo Service
```

Postfix compression would compress out the trailing blanks but the data values after prefix compression would look like this:

```
AA Auto Rentals
2A - California
2B Incorporated
3Towing
1BC Limo Service
```

If the value of *average data length* is much different from the maximum size of the field(s) that make up the index then either there are many fields with many trailing blanks, or the data values are very similar. The index may then be decreasing performance because most of the index and data pages will be read instead of subsets of each.

Multi-Generational Architecture
What Is MGA?
Multi-generational architecture derives it name from process by which InterBase updates a record. Each time a record is updated or deleted a **new copy** (generation) of the record is created. Its main benefit is that writers do not block readers. This means you can run a single query for weeks while people are updating the database. The answer you get from the query will be consistent with the committed contents of the database when you started your query transaction. How does this work?
Every operation on the database, whether it is a read or a write, is time stamped with a **transaction number**. They are assigned sequentially in ascending order. A user who has transaction number 20 started work with the database earlier in time than someone with transaction 21. How much earlier one cannot tell, all you know is that transaction 20 started before transaction 21. And you know what state transaction 20 was in when you started transaction 21. It was either active, committed, rolled back, limbo, or dead. We are just concerned with active, committed, and rolled back for this discussion.
Every record in the database is stamped with the transaction number that inserted, updated, or deleted the record. This number is embedded in the record header. When a record is changed, the old version of the record is kept with the old transaction number and the new version gets the transaction number that changed it. The new version of the record has a pointer to the old version of the record. The old version of the record has a pointer to the prior version of it, and so on. There is a mechanism in place to determine how many old versions need to be kept. If necessary, it will keep every version that has been created.
When you update a record in the database, the old version is compared to the new version to create a **back difference record** (BDR). The BDR is moved to a new location and the new version is written in same the location where the original version was. Even though we keep old versions or records around, the BDR will never be larger than its ancestor. Usually, it will be very small unless you are changing the whole record. With deleted versions it is even smaller. The version being deleted is kept intact as a BDR with the new version just having the current transaction number and a flag indicating that the record is deleted.

Now lets take a look at an added benefit, the ability to **lock** a record without taking out an explicit record lock. Assume that transaction 21 (*t21*) wants to update a record that you are viewing with transaction 20 (*t20*). If *t21* updates the record before you can issue the update, then they have effectively locked the record because the new version of the record will be stamped with transaction number 21. If *t20* tries later, or just a split-second later, the system will immediately detect there is a new version of the record and deny the update. There are simple rules for dealing with transactions and record versions:

- **If your transaction number is less than the record's transaction number, then you cannot see or update it.**
- **If your transaction number is equal to the record's transaction number, then you can see and/or update it.**
- **If your transaction number is greater than the record's transaction number** *AND* **that transaction was committed before you started your transaction, then you can see and/or update it.**

Garbage Collecting

Even with our efficiencies in keeping the BDRs few in number, the database can still accumulate a great deal of unnecessary record versions, i.e. *garbage*. There are two ways to clean out all the garbage from the database. The first is called **Cooperative Garbage Collection**. It happens automatically every time a record is touched, on a select, update, or delete operation. When the record is touched, the InterBase kernel follows the pointers to each BDR and compares its transaction number with what is called the **Oldest Interesting Transaction** (OIT). This number is kept in the header page for the database. If the BDR's transaction number is less than the OIT, then the BDR can be purged from the database and the space reclaimed for new data. This *will not* clean up deleted records and their BDRs.

Sweeping

The second method is called **sweeping** the database. It can either be kicked off manually or automatically. By default, when the OIT is **20,001** transactions less then the *Oldest Active Transaction* number, the process that tried to start the transaction will sweep the entire database and remove as many BDRs as possible. While this is happening, other users can continue to use the database. This threshold can be changed.

If you are going to start the sweep manually then it is advised that you first make sure there is no one connected to the database. This will not only clean out the BDRs and clean out the erased records, but also update the OIT number on the header page to be one less than the Oldest Active Transaction number. It can do this because there are no other active transactions that might need to see any of the BDRs.

For more information on the OIT and sweeping, see from Borland's Web site.

Backup and Restore as Database Maintenance

Periodically you will want to shut down the database and backup and restore it. Note that the backup is performed as a transaction, which means that it sees only a snapshot of the committed records in the database at the time the backup began. This will backup only the current committed version of each record while also putting all the data for each relation on contiguous pages in the database. It will also rebuild all the indices and reset the statistics for each. This usually will increase performance significantly.

Now take this one step further. All of the **metadata** is stored in InterBase tables. This means that it is also multi-generational and has transaction numbers associated with it. If you change the metadata (like a field type) you are actually changing records in an InterBase table. The old versions are kept, and data that used the structure specified by old metadata versions are not changed to match your alteration of the database structure. Because the metadata has old versions, it is possible to have one record with the most current version with one structure and the next record with a different structure. InterBase resolves these via the transaction numbers and the metadata. When you do a query that returns data that is in an old structure, InterBase retrieves the data and must dynamically *convert* the data to the current structure. If the data are in a structure that is three generations old, it goes through three conversions before being returned to you. Do a backup and restore, and finally the data are physically converted to the current structure. All this is done in the name of **performance**. When you commit a transaction, the record versions created by operations during that transaction are already written in the database, so only the status of the transaction has to be updated. Thus, commit and rollback are **fast**. Similarly, when you make a metadata change, InterBase does not change all the data to match the new structure; there may be gigabytes of data to change. Thus, metadata changes are **fast**.

Changing Database Physical Properties

Database Page Size

The database page size determines how much data will be retrieved with one *logical* access from the database. The values permitted by InterBase are 1Kb, 2Kb, 4Kb and 8Kb. The default is 1Kb. A single *logical* access from the database may involve multiple *physical* accesses. For example, on most UNIX systems, the default number of bytes retrieved in a single *physical* access is 512. For a 1Kb page size, two *physical* accesses occur for every *logical* access.

There is a tradeoff then between reading/writing the most data versus physical I/O. The proper page size will be determined by your database requirements. Is the database mostly for reading, update intensive, or a combination? Is accessing BLObS or ARRAYS a priority? What is the record size of your main table, the table that will be accessed most often?
Database page size will also influence the tuning of indices. The section on will review the tool called *GSTAT* that can analyze the layout of an index. If there are too many levels in the index then increasing the page size will reduce the depth of the index structures and usually improve performance. Another rule of thumb is to try to keep the indices on the main table to three or fewer levels.
Another effect of increasing the database page size is to implicitly increase the InterBase .
If you need to change the database's page size after creation, you can do this by doing a backup and then restore the database with a different page size. Page 110 in the *InterBase Client for Windows User's Guide* details how to do this.

Multi-File Databases
The database can also be made up of many different files. This allows you to effectively use the available disk space on multiple volumes. The user always refers to the first file, the *database name*, and never has to know about the secondary files. This also allows a database to be larger than the operating system limit for a single file. This *does not* allow the DBA to specify in which file individual objects in the database may be placed. In other words, you cannot assign *Relation A* to the first file, *Relation B* to the second file and the indices for both to the third file. The files are used and filled in sequential order and can contain data from any table or index in the database. In fact, as data grows over time, the pages used for individual tables or indices are likely to become roughly interleaved.
You can add new files to the database without taking the database offline or interrupting users who are doing work. One reason to do this is if your database is growing and threatens to outgrow the disk volume it resides on. Adding an additional file means that when the primary database file fills up, subsequent data are automatically stored in the secondary file(s). Below is an example of the ISQL command to ad a second file to the database. By doing this, the primary database will top off at 50,000 database pages.

```
ALTER DATABASE ADD FILE "second_file.gdb" STARTING AT 50001;
```

If you need to rearrange the distribution of pages in a multi-file database, you can do it by doing a backup and then restore the database, specifying the secondary files and the attributes. Page 109 in the *InterBase Client for Windows User's Guide* details how to do this.

Database Shadows
Another physical property is the ability to create shadows of the database. Shadows are **carbon copies** of the database, an *exact* duplicate. The main reason for shadows is to protect yourself from hardware failure. First you have to set up one or more shadows on other disk partitions or better still, other machines. If your primary disk or server fails the users can reconnect to the shadow after the DBA has enabled it. This is much quicker than restoring from a backup. Users can be working normally in minutes, rather than hours.
Shadows can also be composed of multiple files just like normal databases. The files comprising a shadow are not required to match the sizes or filenames of the files comprising the master database.
The major drawback to shadows is that they increase the number of writes the database server does. If you have only one shadow then every write is duplicated. If you have two shadows then every write is tripled. If the shadow is located on another machine and the writing is going through NFS, then it takes even longer. There is a tradeoff in this case between I/O performance and data protection.

Changing Database Runtime Properties
Delphi 2.0 included InterBase WI-V4.1.0. This version of InterBase introduced some new controls for tuning performance on the server. These controls will be in future versions of InterBase.

Classic Architecture
Prior to version V4.1.0 of InterBase, the server was designed in a way that is now called the *Classic Architecture*. In this design, each client connection to the server spawns a separate process. Each process does I/O directly to the database files, and negotiates access to the database files by interprocess communication methods like *semaphores*. Each of these server processes also keeps a cache of database pages in its own address space. The cache contains data pages, index pages, BLOb pages, and all other pages that the process reads from the database and may have to write back to the database.
The size of the cache is tunable programmatically, as an entry in the *database parameter block* to the InterBase API function `isc_attach_database`. The size is specified in database pages. By default the size is 75 pages. Increasing the cache can decrease the frequency of I/O operations that are required as the client does database operations. But this is a tradeoff with the memory requirements of the process. 60 processes that each have 300 2Kb database pages use over 35 Mb of memory. At some point, the cache can no longer be kept in physical RAM and it begins swapping

out to virtual memory. When the cache pages are being swapped to the hard disk, the benefit of having a cache at all is defeated. Your InterBase cache size should be tempered with the amount of physical RAM on the machine in mind. Also note that if you increase the database's page size, the actual memory used in the cache is increased proportionally. The cache is configured in number of database pages, not in kilobytes. This principle holds for the Superserver cache, too.

Superserver Architecture
In V4.1.0, InterBase introduces the *Superserver Architecture* (actually, a kind of Superserver was implemented for the NetWare V4.0 version). In this design, all client connections are associated with **one server process**. This server process keeps a database page cache for all clients to share.
The size of the Superserver cache is tunable as a server parameter in the *isc_config* configuration file (in the directory *\Program Files\Borland\IntrBase* by default in WI-V4.1.0). The parameter is `DATABASE_CACHE_PAGES`. The default value for Local InterBase is 256 pages. New database attachments may specify a larger cache size programmatically as with the Classic architecture. These may cause the Superserver cache to expand beyond 256 pages, and this is done as the client attaches. This is part of the IB Settings of the Properties dialog of the WI-V4.1.0 service.

Client Map Size
A memory-mapped file is used for interprocess communication on the server. Each client attachment gets a segment of the file equal to the client map size in bytes. The file is initially 10 times the client map size. Additional clients attaching and detaching will cause the file to grow and shrink by 1 increment of the client map size. This is tunable with the `SERVER_CLIENT_MAPPING` keyword in *isc_config*. This is part of the IB Settings of the Properties dialog of the WI-V4.1.0 service. By default, it is 4096 bytes. This configuration takes effect when the InterBase service restarts.

Priority Class
The superserver process may be given a relative priority class, to allow it to demand more system resources. This only has two options through the InterBase interface, Low (1 in isc_config) or High (2 in *isc_config*). This parameter is specified by the keyword `SERVER_PRIORITY_CLASS` in *isc_config*. This is part of the OS Settings of the Properties dialog of the WI-V4.1.0 service. This configuration takes effect immediately.
NOTE: On Windows NT, the screen saver process does not run at a reduced priority. When running a screen saver that is computationally expensive, such as the OpenGL screen savers demos, all other services running on the NT machine suffer in performance as they compete for CPU resources. Some measurements have indicated that InterBase slows to one tenth of the speed it is capable of. One should disable screen savers on database servers. Or at least configure the screen saver to be a simple black screen.

Working Set
This refers to the set of RAM dedicated for the InterBase process. The minimum working set size specifies the amount of physical RAM that is guaranteed for the InterBase process. The system *may* swap out memory in excess of this figure. The maximum working set size specifies a threshold above which memory used by the InterBase process *will* be swapped out to virtual memory.

NOTE: Working set is only relevant on the Windows NT version of InterBase.
Using the Windows NT Performance Monitor, a database administrator should watch for excessive Page Faults. Raise the minimum working set size in this case. If memory resources permit, set the minimum working set size to at least the amount of cache allocated for the InterBase server. Certainly the maximum should be greater than the size of the InterBase cache, so the cache is not forced to swap.
The default values are zero for both minimum and maximum. This is a special case in which the system determines the working set for the InterBase server process. Both values must be less than the amount of physical RAM on the machine. The minimum must be less than the maximum. These parameters are specified by the keywords SERVER_WORKING_SIZE_MIN and SERVER_WORKING_SIZE_MAX in isc_config. This is part of the OS Settings of the Properties dialog of the WI-V4.1.0 service. This configuration takes effect when the InterBase service is restarted.

Bill Karwin, Senior Technical Support Engineer for Borland InterBase.
bkarwin@interbase.borland.com
This paper was based largely on Paul McGee's Managing InterBase from Borland Developer's Conference 1995.